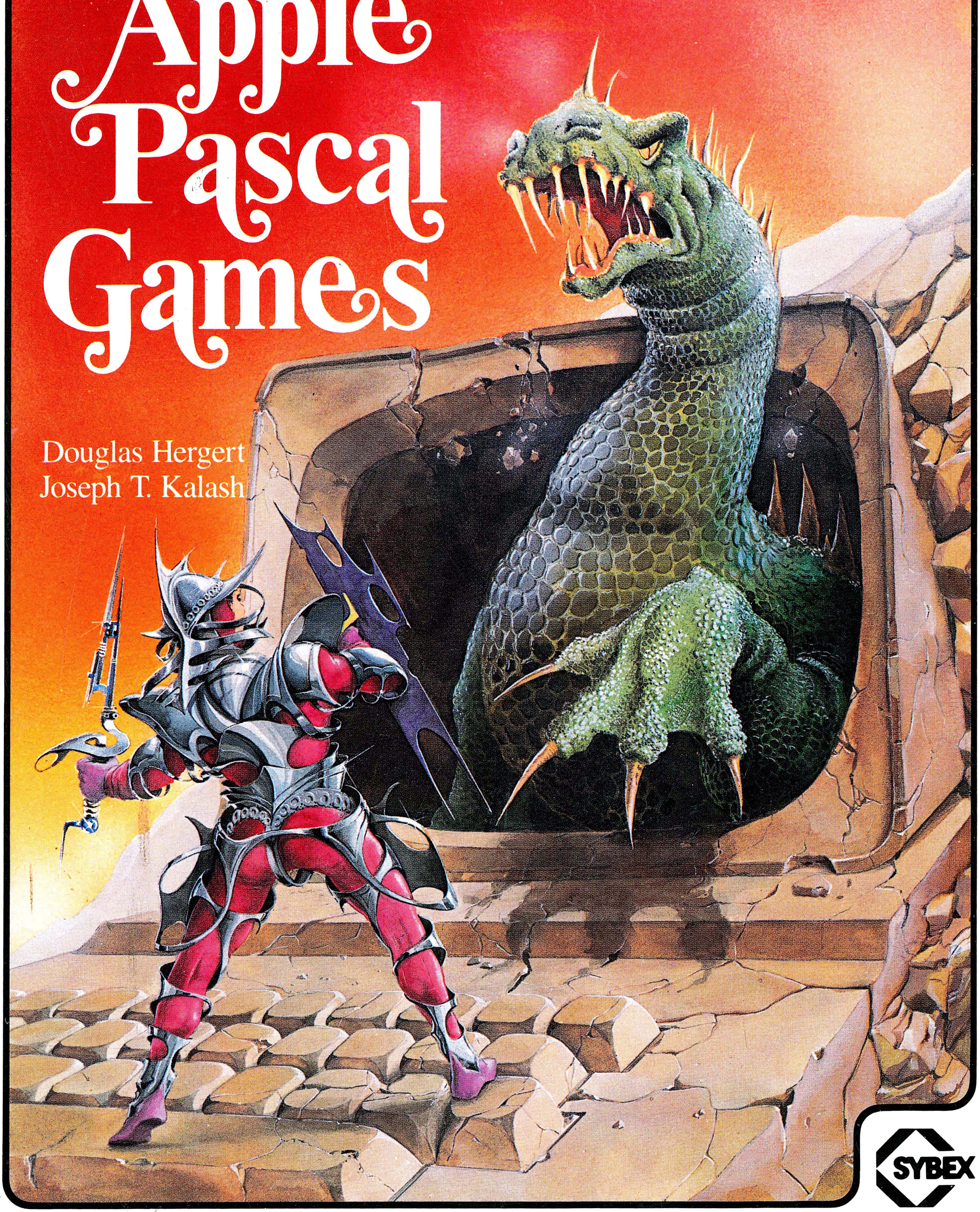
# Apple Pascal Games

Douglas Hergert
Joseph T. Kalash



SYBEX

# APPLE
# PASCAL GAMES

# APPLE
# PASCAL GAMES

DOUGLAS HERGERT
AND JOSEPH T. KALASH

SYBEX®

Berkeley • Paris • Düsseldorf

CREDITS
Cover Design by Daniel Le Noury
Technical Illustrations by j. trujillo smith

Apple is a registered trademark of Apple Computer, Inc.
UCSD Pascal is a trademark of the Regents of the University of California. UCSD Pascal was developed under the direction of Kenneth L. Bowles at the Institute for Information Science, University of California at San Diego.

Every effort has been made to supply complete and accurate information. However, SYBEX assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result.

Copyright © 1981 SYBEX Inc. World Rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 81-16577
ISBN 089588-074-1
First Edition 1981
Printed in the United States of America
10 9 8 7 6 5 4

# Contents

*Introduction* vii

*Acknowledgements* xiii

## *Part I—Simple Games*

GUESSIT 1
KENO 7
OVERUNDER 16
MATHGAME 23
CHUCKALUCK 31
BOGGLE 37
TWINKLE 44
BLOCK 49
NIM 61

## *Part II—More Advanced Games*

HORSERACE 69
KISMET 79
NUMCONVERT 105
ARTILLERY 115
PASCAL 122
BACCARAT 132
DODGE 145
BLACKBOX 169
TELEPHONE 181
SNAKE 191
HANGMAN 203
LIFE 215
BLACKJACK 229
WUMPUS 245
LANDER 261

*Part III—Games that use TURTLEGRAPHICS*

GUNSHOT 283
PICTURE 293

*Part IV—CRIBBAGE*

CRIBBAGE 301

*Appendices*

A—Reserved Words and Functions 350

B—Summary of Pascal 353

# Introduction

> *... The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination ... The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.*
>
> *Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men.*[1]

Thus Frederick P. Brooks describes what he calls "the joys of the craft" of computer programming, in the first chapter of his admirable book, *The Mythical Man-Month.* It is in the spirit of these words that we present *Apple Pascal Games.* We hope that the process of reading and understanding—and perhaps revising—the Pascal programs in this book will delight the reader as much as playing the games themselves. This has been our goal in designing this book.

These days computer games are all around us. Every drug store and corner bar is installing its computerized pinball game; toy manufacturers are busy producing more and more electronic games in portable format; even digital watches with game modes are now appearing on the market. However, all of these game implementations hide what is perhaps the greatest computer game of all—*programming* the computer to "play" a game. Happily, home computers such as the Apple® have now made programming accessible to the games enthusiast. Many games have been written to be played on home computers, and several fine books have been published describing how to write these games in the BASIC language.

Pascal is an elegant, versatile, clearly structured language that is easier to read, and just about as easy to learn as BASIC. The Apple Pascal™ system (which incorporates UCSD Pascal™ plus Apple extensions) includes a simple, yet powerful menu-driven operating system, and an efficient editor. Several software "units" extend the power of Pascal for features that are particularly important to games programs, features such as the RANDOM and KEYPRESS functions, and the TURTLEGRAPHICS procedures.

---

[1] Brooks, Frederick P., Jr., *The Mythical Man-Month: Essays on Software Engineering* (Reading, Mass.: Addison-Wesley, 1978).

Each of the Pascal programs in this book is described in two ways. First, in the sections called "The Game," we explain what the program *does.* These sections require no knowledge of programming, and can serve as a guide for the reader who merely wants to type the programs in and play the games. A sample run is provided for all but the few games that are exclusively screen-oriented.

Then, in the sections called "The Program," we offer an *organization* or *structure chart*[2] for each program, and we explain *how* the program works. The reader should refer frequently to the program listings themselves while reading these sections. We hope the result will be a fairly complete understanding of what is going on in each program.

Figures 1 and 2 illustrate the relationship between a Pascal program and the form of the structure chart that we have used in this book. Figure 1 is simply the "skeleton" of a program—showing the organization of a program, but no actual statements. There are several important points to notice in this program:

- The "main program" block is placed at the very end of a Pascal program. This block controls the action of the program by calling the procedures and functions that are above it.
- Constants and variables (defined in the **CONST**, **TYPE**, and **VAR** sections) may be *global,* i.e., usable throughout the program, or *local,* i.e., meaningful only in the procedure (or *block* of procedures) where they are defined.
- The **USES** statement makes the special "units" available to the program. The APPLESTUFF unit, used in many of the programs in this book, contains RANDOM, RANDOMIZE, and KEYPRESS. The TRANSCEND unit supplies the transcendental functions; it is used in the Artillery game.
- The Pascal comment delimiters (* and *) can be used in a variety of ways to document a program.

---

[2] The structure charts of this book are closely related to the structure charts from the Yourdon School of Structured Design, or the Visual Table of Contents (VTOC) of IBM's HIPO documentation technique.

Figure 2, the structure chart for the program in Figure 1, simply charts the hierarchy of procedures in the program. The main program in this case calls procedures ONE and THREE; procedure ONE, in turn, calls procedure TWO.

```
PROGRAM EXAMPLE;
USES APPLESTUFF, TURTLEGRAPHICS, TRANSCEND;
CONST
      (* GLOBAL CONSTANT DECLARATIONS *);
TYPE
      (* GLOBAL TYPE DECLARATIONS *);
VAR
      (* GLOBAL VARIABLE DECLARATIONS *);
PROCEDURE ONE (      (* FORMAL PARAMETERS OF
                      * PROCEDURE ONE, INCLUDING
                      * VALUE PARAMETERS,
                      *)
           VAR       (* AND VARIABLE PARAMETERS *)
             );
  VAR
      (* LOCAL VARIABLES FOR PROCEDURE ONE *);
  PROCEDURE TWO;     (*
                      * PROCEDURE TWO IS LOCAL
                      * TO PROCEDURE ONE.
                      *)
    BEGIN

      (*
       * MAIN BODY OF PROCEDURE TWO.
       *)
    END;
  BEGIN
      (*
       * MAIN BODY OF PROCEDURE ONE;
       * CONTAINS A CALL TO PROCEDURE TWO.
       *)
  END;
```

*Figure 1: Skeleton of a Pascal Program*

```
FUNCTION THREE (          (*
                           * FORMAL PARAMETERS OF THREE.
                           *)
                ):        (* TYPE OF VALUE THAT THREE RETURNS *);
  BEGIN
    (* BODY OF FUNCTION THREE *)
  END;

BEGIN
    (*
     * MAIN PROGRAM.
     * CONTAINS CALLS TO
     * PROCEDURE ONE AND
     * FUNCTION THREE.
     *)
END.
```

*Figure 1: Skeleton of a Pascal Program (cont.)*

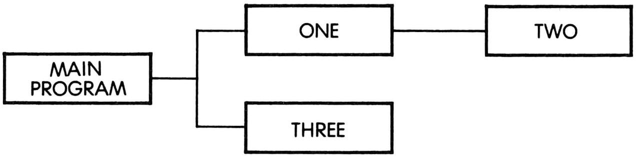


*Figure 2: Structure Chart for Program EXAMPLE*

To accommodate a range of readers—from those who are new to Pascal to those who are already Pascal loyalists—we have organized the games in this book into four progressively more challenging parts.

Part I contains nine rather easy games, some of which have been published many times in BASIC; the programs in this section provide an informal introduction to some of the more elementary features of Pascal. The reader who is already familiar with BASIC versions of these games will be especially interested to see how they are created in Pascal.

Part II provides a collection of more sophisticated game programs. The section is ideal for the reader who wishes to study particular features of Pascal syntax and data structures in the context of significant working programs. The following is a partial list of the features that are discussed and illustrated in the programs of this section:

- Record types — Horserace
- Set types and set operators — Wumpus
- Pointer types and linked lists — Pascal
- Enumerated (or "scalar") types — Snake, Blackbox
- File types and file handling functions — Hangman (and Makedict)
- Recursive procedure calls — Telephone
- String conversion and input validation — Lander
- The **CASE** statement — Kismet

Part III introduces TURTLEGRAPHICS, the Apple graphics package. Most of the major procedures and functions of the package are used in the two programs of this section.

Finally, Part IV presents one long program, Cribbage, which is rich in illustrations of how to use many of the elements of Pascal. Because of its length, Cribbage was stored in three different files to be run on the Apple; the program thus has examples of the include-file compiler option and the FORWARD statement.

The appendices to this book provide a quick summary of Pascal, and can be used as both a learning tool and a reference tool.

While all of these programs were developed and run on the Apple II computer, most of them can be run or easily adapted to run on any UCSD Pascal system.

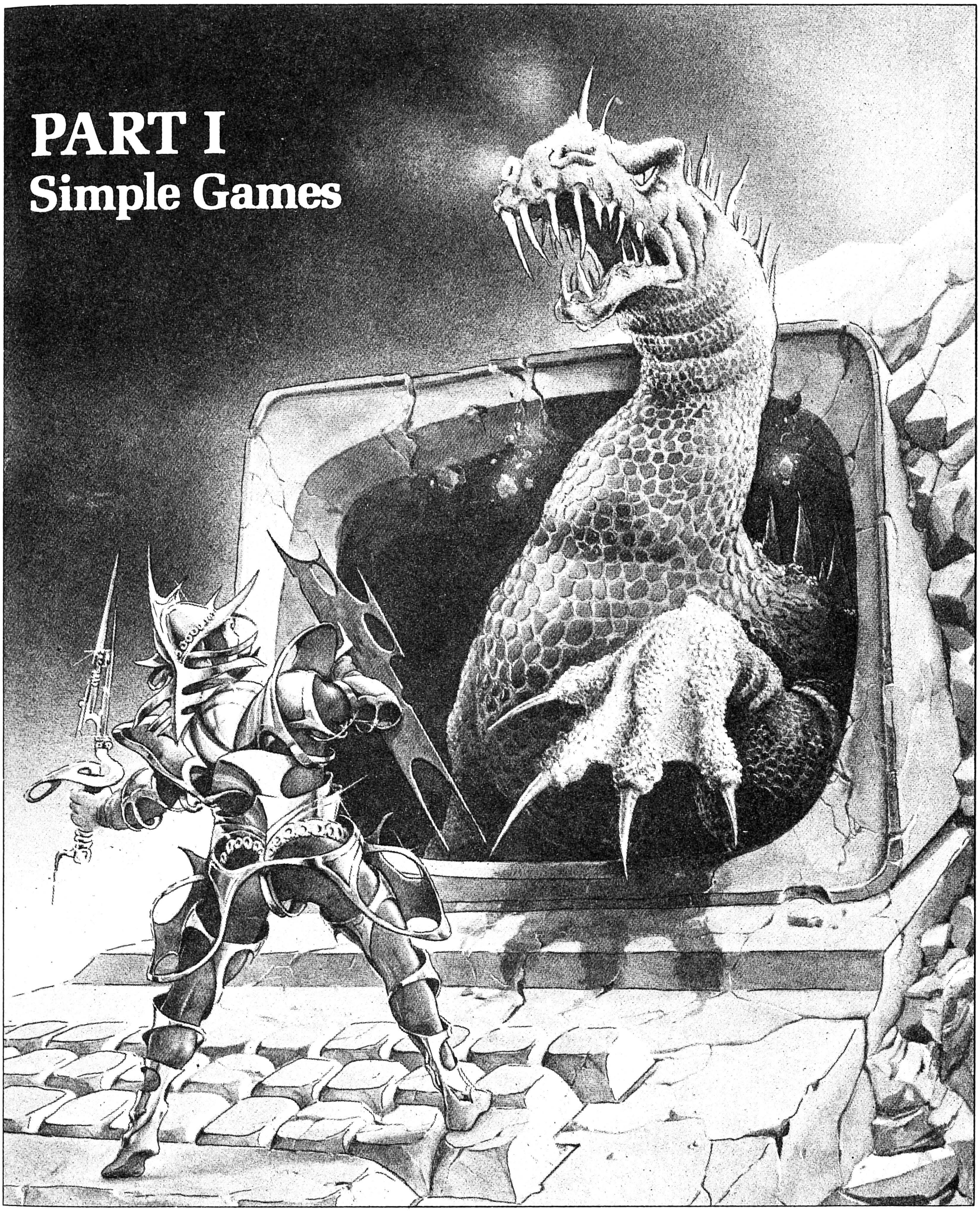
Douglas Hergert
Berkeley, California
August 1981

# Acknowledgements

Many people added to the fun. Charles Koester wrote eight of the programs (Nim, Telephone, Snake, Hangman, Life, Blackjack, Wumpus, and Lander); the remaining programs are by Joseph Kalash. Daniel Le Noury created the cover and the drawings. Rudolph Langer gave sensible and sensitive editorial advice; Jim Compton and Hilda van Genderen conscientiously proofread the galleys. The book was designed and produced by Roger Gottlieb and his staff. The layout and the technical drawings were prepared by j. trujillo smith. Millea Kenin was responsible for pasteup; she was assisted by Judy Wohlfrom and Kristen Harber. Mati Sikk, Gladys Varon, Janet Rampa, and Elaine Foster performed expertly in those ultimate electronic word games—typesetting and word processing.

# PART I

## Simple Games



# GUESSIT

**The Game.** Here is a classic computer guessing game, written in Pascal. The computer chooses a number from 1 to 100, and challenges you to guess the number. After each guess you will receive a message:

```
TOO HIGH
```

or

```
TOO LOW
```

or, if you guess correctly:

```
CORRECT!!!!
```

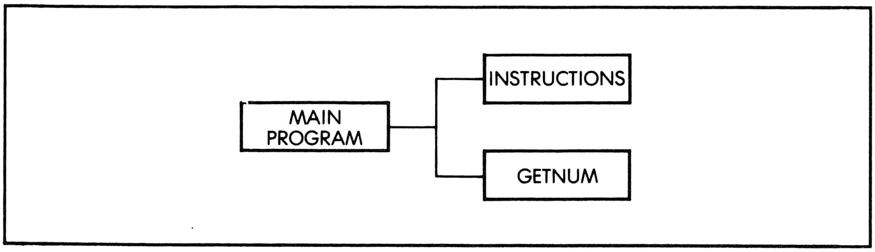
You have six chances to guess the right number; if you haven't guessed it on the sixth try, the number will be revealed to you. You can keep playing this game as long as you want, without restarting the program. When you want to quit, answer the question:

```
WANT TO TRY AGAIN?
```

with an N.

```
THIS IS MATCH IT

I WILL CHOOSE A NUMBER BETWEEN 1 AND 100
YOU WILL TRY TO GUESS THAT NUMBER
IF YOU GUESS WRONG, I WILL TELL YOU IF YOU GUESSED
TOO HIGH, OR TOO LOW
YOU HAVE 6 TRIES TO GET THE NUMBER

ENJOY

YOUR GUESS? 50
TOO HIGH
YOUR GUESS? 25
TOO HIGH
YOUR GUESS? 12
TOO LOW
YOUR GUESS? 18
CORRECT!!!!


WANT TO TRY AGAIN? Y
YOUR GUESS? 50
TOO HIGH
YOUR GUESS? 25
TOO LOW
YOUR GUESS? 37
TOO HIGH
YOUR GUESS? 32
TOO LOW
YOUR GUESS? 35
CORRECT!!!!


WANT TO TRY AGAIN? N
```

# The Program.



This program uses the built-in procedure RANDOM to "choose" a number between 1 and 100; thus the main program begins with a call to the Pascal procedure RANDOMIZE. (RANDOMIZE insures an unpredictable sequence of random numbers.) The main program calls procedure INSTRUCTIONS to print out a description of the game and then enters a **WHILE** loop, which retains control of the program until the player types an 'N' at the end of a game:

```
WHILE CH <> "N" DO ...
```

GUESS and NUMTRY are initialized to zero at the beginning of the **WHILE** loop. NUMTRY is the counter for the number of guesses that have been made. It is incremented (after each call to procedure GETNUM) with the statement:

```
NUMTRY := NUMTRY + 1
```

The number to be guessed is chosen by the following statement:

```
NUMBER := (RANDOM MOD MAXNUM) + MINNUM
```

Each call to RANDOM returns an integer between 0 and 32767. The **MOD** operation supplies the *remainder* from the division of two integers. Thus, RANDOM **MOD** MAXNUM, where MAXNUM has the value 100, will return an integer from 0 to 99. Adding 1 (the value of MINNUM) to this expression adjusts the range to give a number from 1 to 100.

Once the random number is chosen, the player's guesses can be accepted. Procedure GETNUM accepts the guesses. GETNUM is called inside another **WHILE** loop. The **WHILE** statement contains two conditions:

```
(NUMTRY < MAXTRIES) AND (GUESS <> NUMBER)
```

Thus GETNUM is called repeatedly until either the player has used up the six tries, or the correct number has been guessed.

Procedure GETNUM is perhaps the most complicated part of this program. The difficult part is the validation of the input. This validation is performed by the statement:

```
GET(INPUT);
```

and the use of the implicitly defined *buffer variable,* INPUT ^. The first character of the input is assigned to INPUT ^. Then, the following statement tests the character to see if it is **IN** the set of integers from 1 to 9:

```
IF NOT (INPUT ^ IN ['1'..'9']) THEN . . .
```

An *illegal* input (that is, anything other than a digit from 1 to 9) results in an error message:

```
ILLEGAL NUMBER
```

and a *recursive call* to GETNUM after a READLN to throw away whatever might remain on the line with the illegal input. This means that if the input is illegal, GETNUM starts again from the top (and asks for a new guess). If the input is *legal,* on the other hand, the number is read into the variable GUESS:

```
READLN(GUESS);
```

Another validation test remains, however. GUESS must be less than or equal to MAXNUM. If it is not, that is:

```
IF GUESS > MAXNUM . . .
```

then an error message is displayed, and once again GETNUM is called recursively.

Finally, once the value of GUESS is validated, the program must inform the player if the guess is too low, too high, or correct. This is done by a series of nested **IF/THEN/ELSE** statements at the end of procedure GETNUM.

Back in the main program, NUMTRY is incremented for the new guess. When all the tries are used up, and a correct guess has not been made, the program displays the correct number and asks the player:

```
WANT TO TRY AGAIN?
```

```
(*
 * GUESS IT
 * A COMPUTER GUESSING GAME
 *)
PROGRAM GUESSIT (INPUT,OUTPUT);
USES   APPLESTUFF;
CONST
   MAXTRYS      =  6;          (* USER ALLOWED 6 TRIES TO GUESS NUMBER *)
   MAXNUM       = 100;         (* MAXIMUM NUMBER *)
   MINNUM       =  1;          (* MINIMUM NUMBER *)

VAR
   CH : CHAR;                  (* USED TO ANSWER QUESTIONS *)
   GUESS : INTEGER;            (* HUMAN'S GUESS *)
   NUMBER: MINNUM..MAXNUM;     (* THE NUMBER ITSELF *)
   NUMTRY: 0..MAXTRYS;         (* NUMBER OF TIMES HUMAN HAS GUESSED *)


   (*
    * INSTRUCTIONS
    * PRINTS OUT THE INSTRUCTIONS
    *)
   PROCEDURE INSTRUCTIONS;
   BEGIN
      WRITELN('THIS IS MATCH IT');
      WRITELN;
      WRITELN('I WILL CHOOSE A NUMBER BETWEEN 1 AND 100');
      WRITELN('YOU WILL TRY TO GUESS THAT NUMBER');
      WRITELN('IF YOU GUESS WRONG, I WILL TELL YOU IF YOU GUESSED');
      WRITELN('TOO HIGH, OR TOO LOW');
      WRITELN('YOU HAVE ',MAXTRYS,' TRY TO GET THE NUMBER');
      WRITELN;
      WRITELN('ENJOY');
      WRITELN
   END; (* INSTRUCTIONS *)


   (*
    * GETNUM
    * GET A GUESS FROM THE HUMAN,
    * WITH ERROR CHECKING
    *)
   PROCEDURE GETNUM;
   BEGIN
      WRITE('YOUR GUESS? ');

      (*
       * WAIT FOR A CHARACTER
       *)

      GET(INPUT);
      (*
       * CHECK TO SEE IF THE CHARACTER
       * COULD BE A NUMBER
       *)
      IF NOT (INPUT^ IN ['1'..'9']) THEN
      BEGIN
         READLN(INPUT);
         WRITELN('ILLEGAL NUMBER');
         GETNUM                 (* RECURSE FOR LEGAL INPUT *)
      END
      ELSE
      BEGIN

         READLN(GUESS);
         IF GUESS > MAXNUM THEN
         BEGIN
            WRITELN('ILLEGAL NUMBER');
            GETNUM
         END
         ELSE
         BEGIN
            IF GUESS > NUMBER THEN
               WRITELN('TOO HIGH')
            ELSE
               IF GUESS < NUMBER THEN
                  WRITELN('TOO LOW')
               ELSE
                  WRITELN('CORRECT!!!!')
         END
      END
   END; (* GETNUM *)

(*
 * MAIN
 *)

BEGIN
   INSTRUCTIONS;
   RANDOMIZE;
   WHILE CH <> 'N' DO
   BEGIN
      GUESS := 0;
      NUMTRY := 0;
      NUMBER := RANDOM MOD MAXNUM + MINNUM;     (* GET NUMBER *)
      WHILE ( NUMTRY < MAXTRYS ) AND ( GUESS <> NUMBER ) DO
      BEGIN
         GETNUM;
         NUMTRY := NUMTRY +1
      END;
      WRITELN;
      IF GUESS <> NUMBER THEN
         WRITELN('THE NUMBER WAS ',NUMBER);
      WRITELN;
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
   END
END.
```

# KENO

**The Game.** Welcome to the Keno lounge. The atmosphere is comfortable (even if a bit gaudy), the drinks are on the house, and you can play a game of Keno for only $1.20.

The rules are simple. All you do is pick eight lucky numbers—"spots"—from 1 to 80. Meanwhile, the house picks 20 spots (also from 1 to 80) at random. You are a winner if 5 or more of your spots match the house's spots. What could be easier?

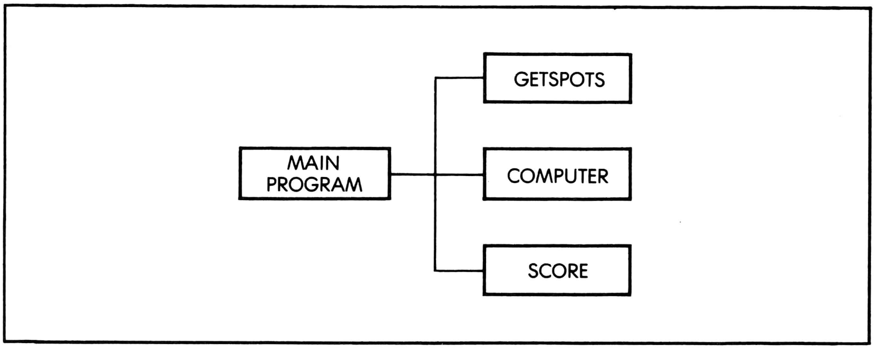
Here is what you can win:

For 5 matches—$10
For 6 matches—$100
For 7 matches—$2,200
For 8 matches—$25,000!!!

Play as many games as you want. The house (that is, the computer) will keep track of your winnings, and will be delighted to see you spend your entire afternoon here in the Keno lounge. Think of it this way: the longer you stay, the more chances you have for the big win!

```
YOUR SPOT? 12
YOUR SPOT? 16
YOUR SPOT? 23
YOUR SPOT? 27
YOUR SPOT? 35
YOUR SPOT? 44
YOUR SPOT? 65
YOUR SPOT? 77
THE COMPUTER CHOOSES
  4  6  8 14 15 20 26 28 33 34
 39 43 46 57 69 70 74 76 77 80
SORRY, BUT YOU ONLY MATCHED 1

SO FAR YOU HAVE LOST $ 1.20


WANT TO PLAY KENO AGAIN? Y
YOUR SPOT? 27
YOUR SPOT? 33
YOUR SPOT? 78
YOUR SPOT? 17
YOUR SPOT? 22
YOUR SPOT? 19
YOUR SPOT? 55
YOUR SPOT? 23
THE COMPUTER CHOOSES
  1 15 17 19 22 23 26 27 30 33
 37 38 48 51 52 60 62 63 68 80
YOU LUCKY PERSON, YOU HAVE 6 MATCHES
THAT MEANS YOU HAVE MADE 100

YOUR TOTAL MONEY IS $ 97.60


WANT TO PLAY KENO AGAIN? Y
YOUR SPOT? 34
YOUR SPOT? 25
YOUR SPOT? 27
YOUR SPOT? 78
YOUR SPOT? 66
YOUR SPOT? 15
YOUR SPOT? 32
YOUR SPOT? 66
YOU HAVE ALREADY CHOSEN SPOT 66
YOUR SPOT? 65
THE COMPUTER CHOOSES
  4  8 10 17 18 22 23 26 30 36
 37 38 49 55 57 62 66 67 77 80
SORRY, BUT YOU ONLY MATCHED 1

YOUR TOTAL MONEY IS $ 96.40


WANT TO PLAY KENO AGAIN? N

ALL RIGHT THEN LEAVE, SEE IF I CARE!!!
YOU MADE $ 96.40
```

# The Program.



This is an excellent program for studying the use of sets in Pascal. The first things to notice are the **TYPE** and **VAR** declarations at the beginning of the program:

```
TYPE
  SPOTTYPE = SET OF 1..TOTALSPOTS;
VAR
  COMPSPOTS: SPOTTYPE;
  PLAYERSPOTS: SPOTTYPE;
  ...
```

Both COMPSPOTS (the spots chosen by the computer) and PLAYERSPOTS (the spots chosen by the player) are declared as set types. Other data structures could have been used (a BASIC version of this program would store the spots in two one-dimensional arrays) but, as we will see, the use of sets is the clearest and most elegant way to *compare* the computer's spots with the player's spots.

The main program initializes MONEY to zero. (The value of MONEY will thus be negative until the player wins, since COST—the charge for a game—is subtracted from MONEY for every game that is played.)

Most of the rest of the main program is taken up by a **WHILE** loop that contains the procedure calls. A complete game consists of one call each to GETSPOTS, COMPUTER, and SCORE. The procedures are called continually until the player types an N in response to the question:

```
WANT TO PLAY KENO AGAIN?
```

Procedure GETSPOTS asks for the player's spots, and stores them in the set PLAYERSPOTS. PLAYERSPOTS is initialized as an empty set:

```
PLAYERSPOTS := [ ]
```

Each value input for SPOT goes through two validation tests before it is stored in PLAYERSPOTS. First, the value must be in the correct range (1 to 80):

```
IF (SPOT < 1) OR (SPOT > TOTALSPOTS) THEN
```

Second, the value must be a new one; that is, it must be a spot that the player has not already chosen in the present game. This test contains the first of several uses in this program of the relational operator **IN**:

```
IF SPOT IN PLAYERSPOTS THEN . . .
```

If the SPOT value passes both of these tests, then it is added to the set:

```
PLAYERSPOTS := PLAYERSPOTS + [SPOT]
```

Procedure COMPUTER contains a similar sequence of statements to create the set COMPSPOTS. The main difference here is that each value of SPOT is calculated from the value returned by a call to RANDOM:

```
SPOT := 1 + (RANDOM MOD TOTALSPOTS)
```

RANDOM **MOD** TOTALSPOTS produces an integer from 0 to 79. Adding 1 to this value gives an integer from 1 to 80.

The second part of procedure COMPUTER is the one part of this program that would be simpler if COMPSPOTS were an array rather than a set. This routine must assign values from the whole range of SPOTTYPE (1 to 80) to the variable SPOT and then WRITE any value that satisfies the condition:

```
IF SPOT IN COMPSPOTS . . .
```

(Note that there is no direct way to access and write the elements of a set.) If COMPSPOTS were an array, on the other hand, the routine could simply write all twenty values of the array, using a **FOR** loop:

```
FOR I := 1 TO NUMCOMPSPOTS DO
  BEGIN
    WRITE (COMPSPOTS[I]: 3);
    IF (I = 10) THEN WRITELN
  END;
```

However, the slight programming inefficiency here is more than compensated by the ease of comparing the sets COMPSPOTS and PLAYERSPOTS later in procedure SCORE.

Procedure SCORE begins by deducting the COST of a game from the player's present assets, MONEY, and then begins a **FOR** loop to compare the player's spots with the computer's spots. For values of SPOT from 1 to 80, if SPOT is **IN** both COMPSPOTS and PLAYERSPOTS, then I (the match counter) is incremented by 1. If the final value of I represents a winning number of matches, then a **CASE** statement is used to add the appropriate amount of PAYOFF money to the player's assets. (The PAYOFF amounts were set in the **CONST** statement at the beginning of the program.)

In this final procedure, if COMPSPOTS and PLAYERSPOTS had been arrays, the comparison would have been more difficult. It would have involved nested **FOR** loops, and would have read less clearly:

```
FOR J := 1 TO NUMPLAYERSPOTS DO
  FOR K := 1 TO NUMCOMPSPOTS DO
    IF PLAYERSPOTS[J] = COMPSPOTS[K]
      THEN I := I + 1;
```

All in all, this program illustrates a situation in which the use of sets leads to easy writing and easy reading of each procedure.

```
PROGRAM KENO (INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   PAYOFF5         = 10;    (* FOR 5 SPOTS PLAYER GETS $10 *)
   PAYOFF6         = 100;   (* FOR 6 SPOTS PLAYER GETS $100 *)
   PAYOFF7         = 2200;  (* FOR 7 SPOTS PLAYER GETS $2200 *)
   PAYOFF8         = 25000;(* FOR 8 SPOTS PLAYER GETS $25000 *)
   TOTALSPOTS      = 80;    (* TOTAL OF 80 POSSIBLE SPOTS *)
   NUMCOMPSPOTS    = 20;    (* COMPUTER PICKS 20 SPOTS *)
   NUMPLAYERSPOTS  = 8;     (* PLAYER GETS 8 SPOTS *)
   COST            = 1.20;  (* COST TO PLAY A GAME *)

TYPE
   SPOTTYPE = SET OF 1..TOTALSPOTS;

VAR
   COMPSPOTS    : SPOTTYPE; (* SPOTS THE COMPUTER CHOOSES *)
   PLAYERSPOTS  : SPOTTYPE; (* PLAYER'S SPOTS *)
   MONEY        : REAL;     (* PLAYER HAS THIS MUCH MONEY *)
   CH           : CHAR;     (* INPUT CHAR TO SEE IF USER WISHES TO CONTINUE *)

(*
 * GETSPOTS
 * GET THE PLAYER'S 8 SPOTS FROM HIM
 *)

PROCEDURE GETSPOTS;
VAR
   I : 0..NUMPLAYERSPOTS;  (* NUMBER OF SPOTS PLAYER HAS CHOSEN SO FAR *)
   SPOT : INTEGER;         (* NUMBER JUST CHOSEN *)
BEGIN
   I := 0;
   PLAYERSPOTS := [];
   WHILE I < NUMPLAYERSPOTS DO
   BEGIN
      RANDOMIZE;
      WRITE('YOUR SPOT? ');
      READLN(SPOT);
      IF (SPOT < 1) OR (SPOT > TOTALSPOTS) THEN
         WRITELN('ILLEGAL SPOT NUMBER')
      ELSE
         IF SPOT IN PLAYERSPOTS THEN
            WRITELN('YOU HAVE ALREADY CHOSEN SPOT ',SPOT)
         ELSE
         BEGIN
            I := I+1;
            PLAYERSPOTS := PLAYERSPOTS +[SPOT]
         END
   END
END;
```

```
(*
 * COMPUTER
 * HAVE THE COMPUTER PICK ITS SPOTS
 *)

PROCEDURE COMPUTER;
VAR
   I : 0..NUMCOMPSPOTS;    (* NUMBER OF SPOTS CHOSEN SO FAR *)
   SPOT : 0..TOTALSPOTS ;

BEGIN
   I := 0;
   COMPSPOTS := [];
   WRITELN('THE COMPUTER CHOOSES ');
   WHILE I < NUMCOMPSPOTS DO
   BEGIN
      SPOT := 1 + (RANDOM MOD TOTALSPOTS);
      IF NOT (SPOT IN COMPSPOTS) THEN
      BEGIN
         COMPSPOTS := COMPSPOTS + [SPOT];
         I := I+1;
      END
   END;


  (*
   * PRINT OUT THE COMPUTER'S SPOTS IN
   * ORDER BY GOING LINEARLY THROUGH
   * ALL NUMBERS
   *)
  I := 0;
  SPOT := 0;
  WHILE I < NUMCOMPSPOTS DO
  BEGIN
     SPOT := SPOT + 1;
     IF SPOT IN COMPSPOTS THEN
     BEGIN
        WRITE(SPOT:3);
        I := I +1;
        IF I = 10 THEN
           WRITELN
      END;
   END;
   WRITELN
END;

(*
 * SCORE
 * FIND OUT HOW MUCH (IF ANYTHING) THE
 * PLAYER HAS WON
 *)
```

```
PROCEDURE SCORE;
VAR
   I : 0..NUMPLAYERSPOTS;         (* NUMBER OF MATCHES MADE *)
   SPOT : 1..TOTALSPOTS;          (* CURRENT SPOT NUMBER *)
BEGIN
   MONEY := MONEY - COST;         (* CHARGE FOR THE CARD *)
   I := 0;
   FOR SPOT := 1 TO TOTALSPOTS DO
      IF (SPOT IN COMPSPOTS) AND (SPOT IN PLAYERSPOTS) THEN
         I := I+1;
   IF I IN [5,6,7,8] THEN
   BEGIN
      WRITE('YOU LUCKY PERSON, YOU HAVE ',I);
      WRITELN(' MATCHES');
      WRITE('THAT MEANS YOU HAVE MADE ');
      CASE I OF
         5:
            BEGIN
               WRITELN(PAYOFF5);
               MONEY := PAYOFF5 + MONEY
            END;

         6:
            BEGIN
               WRITELN(PAYOFF6);
               MONEY := PAYOFF6 + MONEY
            END;
         7:
            BEGIN
               WRITELN(PAYOFF7);
               MONEY := PAYOFF7 + MONEY
            END;
         8:
            BEGIN
               WRITELN(PAYOFF8);
               MONEY := PAYOFF8 + MONEY
            END
      END
   END
   ELSE
      WRITELN('SORRY, BUT YOU ONLY MATCHED ',I);

   WRITELN;
   IF MONEY >= 0 THEN
     WRITELN('YOUR TOTAL MONEY IS $',MONEY:4:2)
   ELSE
     WRITELN('SO FAR YOU HAVE LOST $',ABS(MONEY):4:2);
   WRITELN
END;
```

```
BEGIN
   MONEY := 0;
   WHILE CH <> 'N' DO
   BEGIN
      GETSPOTS;
      COMPUTER;
      SCORE;
      WRITELN;
      WRITE('WANT TO PLAY KENO AGAIN? ');
      READLN(CH)
   END;
   WRITELN;
   WRITELN('ALL RIGHT THEN LEAVE, SEE IF I CARE!!!');
   IF (MONEY >= 0) THEN
      WRITELN('YOU MADE $',MONEY:5:2)
   ELSE
      WRITELN('YOU LOST $',ABS(MONEY):5:2)
END. (* KENO, 7/5/81 D.H. *)
```

# OVERUNDER

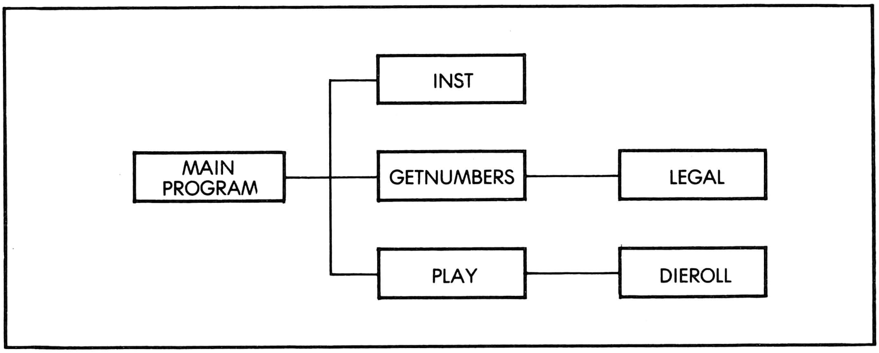
**The Game.** The player chooses a number between 2 and 12 and places a bet. Two dice are rolled. If the number chosen is the same as the roll of the dice, the player wins four times the amount of the bet. Failing an exact match between the "lucky" number and the throw of the dice, the player can still win the amount of the bet if the number and the roll of the dice are both under or both over 7. Otherwise, the amount of the bet is lost.

You begin this game with $500 in your pocket, and you can continue playing as long as your money and stamina hold out.

```
DO YOU WANT INSTRUCTIONS? Y
OVERUNDER
A SIMPLE DICE GAME.
YOU CHOOSE A NUMBER BETWEEN 2 AND 12.
IF THE SUM OF TWO DICE ROLLED IS THE SAME
AS THE NUMBER YOU PICKED, YOU WIN FOUR
TIMES YOUR BET. IF YOUR NUMBER, AND THE
DICE SUM ARE EITHER BOTH UNDER, OR BOTH
OVER 7, THEN YOU WIN THE AMOUNT YOU BET.



WHAT NUMBER DO YOU WANT? 8
YOUR BET? 50

DIE#1    DIE#2    SUM     YOUR#    ROLL
  5        5      10       8       OVER
YOU MADE EVEN MONEY
YOU WON $50
YOUR NEW TOTAL IS $550

WANT TO TRY AGAIN? Y
```

```
WHAT NUMBER DO YOU WANT? 5
YOUR BET? 100

DIE#1    DIE#2    SUM     YOUR#    ROLL
   2       2       4       5       UNDER
YOU MADE EVEN MONEY
YOU WON $100
YOUR NEW TOTAL IS $650

WANT TO TRY AGAIN? Y

WHAT NUMBER DO YOU WANT? 11
YOUR BET? 200

DIE#1    DIE#2    SUM     YOUR#    ROLL
   2       1       3      11       UNDER
YOU LOST $200
YOUR NEW TOTAL IS $450

WANT TO TRY AGAIN? N
```

# The Program.



The main program starts out generously enough by giving the player $500 (the value of STARTMONEY) and offering instructions for the game. Procedure INST, which contains the instructions, is only called if the player types a Y in response to "DO YOU WANT INSTRUCTIONS?" Otherwise, the main program enters a **REPEAT/UNTIL** loop that keeps calling procedures GETNUMBERS and

PLAY as long as the player enters a Y indicating that he wishes to continue.

Procedure GETNUMBERS requests two numbers as input: (1) the number that the player wants to bet on (this value is assigned to the global variable HUMNUM), and (2) the amount of the bet (assigned to BET). GETNUMBERS calls function LEGAL twice in order to receive and validate each input value. Notice the formal parameters of LEGAL: QUESTION, which is of the "dynamic" string type, and MIN and MAX, which are integers. For each call to LEGAL the appropriate prompt question is sent, along with the minimum and maximum for the input values. LEGAL contains two validation tests. The first one, contained in the **WHILE** statement, is to check that the input value is within the proper range. The second test is to make sure the player does not accidentally enter a non-numeric character. This test is implemented with the GET procedure and the buffer variable INPUT^.

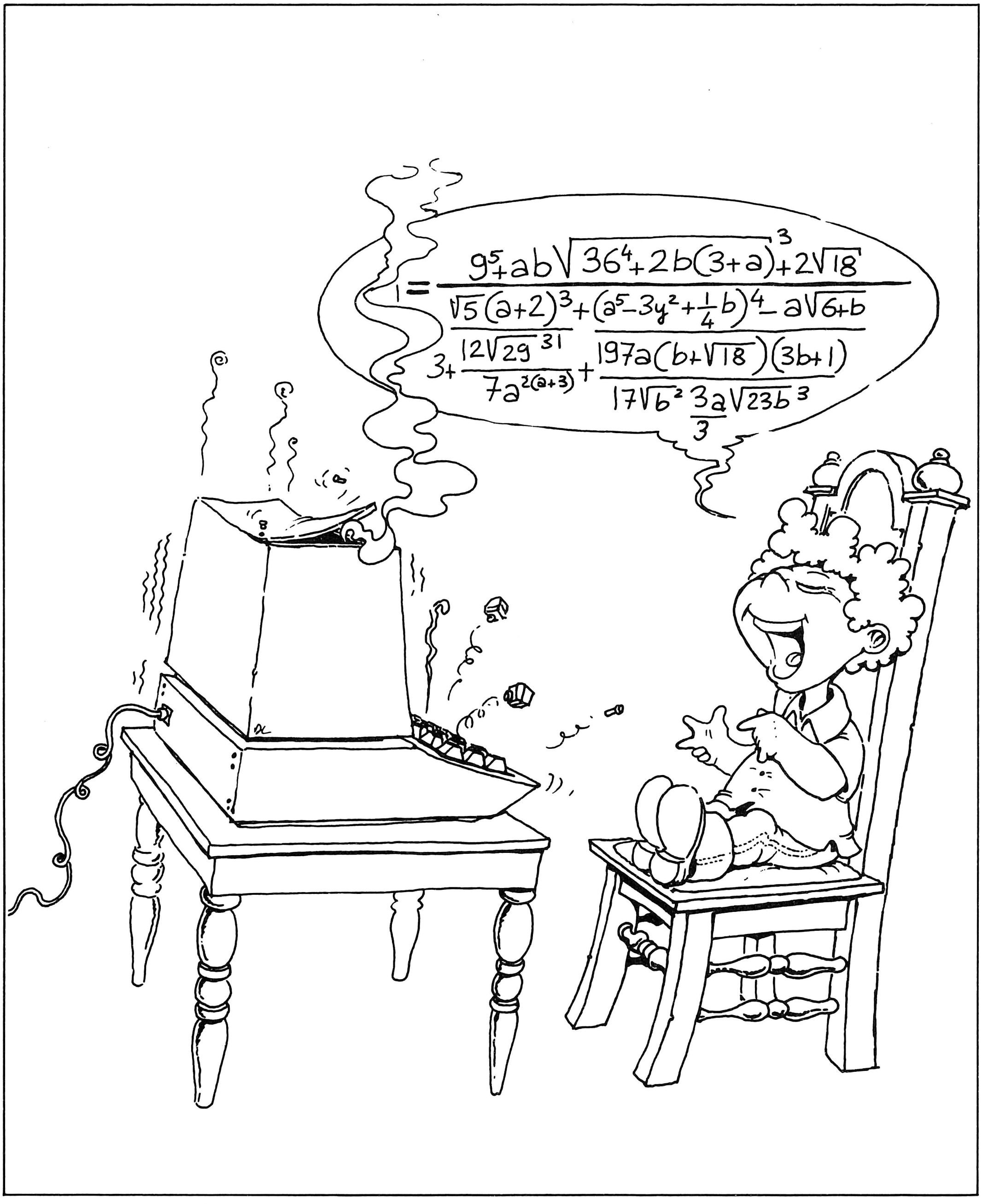
Procedure PLAY calls function DIEROLL twice and adds up the two returned values in SUM to simulate the roll of two dice. It then prints out the value of the throw along with a series of messages indicating the results of the play. The appropriate amount (either BET or 4 * BET) is added to or subtracted from MONEY to arrive at the new total for the player's assets.

Back in the main program, the **REPEAT/UNTIL** block keeps looping until the bleary-eyed player, tired and nearly broke, plucks up the courage to enter an N in response to the insidious question:

```
WANT TO TRY AGAIN?
```

```
PROGRAM OVERUNDER(INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   MAXDIE       =  6;           (* THE MAXIMUM # ON A DIE *)
   MINDIE       =  1;           (* THE MINIMUM # ON A DIE *)

   MAXSUM       = 12;           (* MAXIMUM  SUM FOR TWO DICE *)
   MINSUM       =  2;           (* MINIMUM   "   "   "   "   *)

   STARTMONEY   = 500;          (* AMOUNT OF MONEY PLAYER STARTS WITH *)

TYPE
   DIESUM       = MINDIE..MAXSUM; (* TYPE FOR THE SUM OF DICE *)
   DIETYPE      = MINDIE..MAXDIE; (* A DIE VALUE *)

VAR
   MONEY        : INTEGER;      (* HOW MUCH MONEY PLAYER HAS *)
   BET          : INTEGER;      (* HOW MUCH IS BEING BET THIS TIME *)
   DIE          : DIETYPE;      (* VALUE OF A DIE *)
   SUM          : DIESUM;       (* SUM OF THE 2 DICE *)
   HUMNUM       : DIESUM;       (* NUMBER PICKED BY USER *)
   CH           : CHAR;         (* USED TO ANSWER QUESTIONS *)


   (*
    * PLAY
    * THROWS THE DICE, PRINTS THEM OUT,
    * AND KEEPS SCORE
    *)
   PROCEDURE PLAY;

      (*
       * DIEROLL
       * RETURNS A RANDOM DIE VALUE
       *)
      FUNCTION DIEROLL:DIETYPE;
      BEGIN
         DIEROLL := 1 + RANDOM MOD MAXDIE
      END; (* DIEROLL *)

   BEGIN
      WRITELN('DIE#1    DIE#2    SUM     YOUR#    ROLL');
      DIE := DIEROLL;
      WRITE(DIE:4,'    ');
      SUM := DIE;
      DIE := DIEROLL;
      SUM := SUM + DIE;
      WRITE(DIE:4,'    ',SUM:4,'    ',HUMNUM:4,'        ');
      IF SUM < 7 THEN
         WRITELN('UNDER')
      ELSE
         IF SUM > 7 THEN
            WRITELN('OVER')
         ELSE
            WRITELN('EVEN');
   IF SUM = HUMNUM THEN
   BEGIN
      WRITELN('YOU MATCHED!!!!');
      WRITELN('YOU GET $',BET*4);
      MONEY := MONEY + (BET *4)
   END
   ELSE
      IF ((SUM < 7) AND (HUMNUM < 7)) OR ((SUM>7) AND (HUMNUM>7)) THEN
      BEGIN
         WRITELN('YOU MADE EVEN MONEY');
         WRITELN('YOU WON $',BET);
         MONEY := MONEY + BET
      END
      ELSE
      BEGIN
         WRITELN('YOU LOST $',BET);
         MONEY := MONEY - BET
      END;

   WRITELN('YOUR NEW TOTAL IS $',MONEY)
END; (* PLAY *)

(*
 * GETNUMBERS
 * GET THE GUESS, AND THE BET FROM THE
 * USER
 *)
PROCEDURE GETNUMBERS;
   (*
    * LEGAL
    * GETS A NUMBER BETWEEN MIN AND
    * MAX FROM THE USER, WITH PROPER ERROR
    * CHECKING
    *)
   FUNCTION LEGAL(QUESTION : STRING;MIN,MAX:INTEGER):INTEGER;
   VAR
      NUM : INTEGER;         (* INPUT NUMBER *)
   BEGIN
      NUM := -1;

      WHILE ( NUM < MIN) OR (NUM > MAX ) DO
      BEGIN
         WRITE(QUESTION);
         RANDOMIZE;
         GET(INPUT);         (* SET INPUT^ *)
         WHILE NOT (INPUT^ IN ['1'..'9']) DO
         BEGIN
            READLN(INPUT);   (* SNARF GARBAGE *)
            WRITELN('SORRY, BUT ONLY NUMBERS ARE ALLOWED');
            WRITE(QUESTION);
            GET(INPUT)
         END;
         READLN(NUM);
         IF (NUM < MIN) OR (NUM > MAX) THEN
            WRITELN('SORRY, ONLY NUMBERS BETWEEN ',MIN,' AND ',MAX)
      END;
      LEGAL := NUM
   END; (* LEGAL *)
```

```
BEGIN
   HUMNUM := LEGAL('WHAT NUMBER DO YOU WANT? ',MINSUM,MAXSUM);
   BET := LEGAL('YOUR BET? ',1,MONEY);
END; (* GETNUMBERS *)


   (*
    * INST
    * PRINT OUT A LIST OF INSTRUCTIONS
    *)
   PROCEDURE INST;
   BEGIN
      WRITELN('OVERUNDER');
      WRITELN('A SIMPLE DICE GAME.');
      WRITELN('YOU CHOOSE A NUMBER BETWEEN 2 AND 12.');
      WRITELN('IF THE SUM OF TWO DICE ROLLED IS THE SAME');
      WRITELN('AS THE NUMBER YOU PICKED, YOU WIN FOUR ');
      WRITELN('TIMES YOUR BET. IF YOUR NUMBER, AND THE ');
      WRITELN('DICE SUM ARE EITHER BOTH UNDER, OR BOTH');
      WRITELN('OVER 7, THEN YOU WIN THE AMOUNT YOU BET.');
      WRITELN;
      WRITELN
   END;(* INST *)


BEGIN
   MONEY := STARTMONEY;
   WRITE('DO YOU WANT INSTRUCTIONS? ');
   READLN(CH);
   IF CH = 'Y' THEN
      INST;
   REPEAT
      WRITELN;
      GETNUMBERS;
      WRITELN;
      PLAY;
      WRITELN;
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
   UNTIL ( CH <> 'Y' ) OR (MONEY <= 0);
   IF (MONEY <= 0) THEN
      WRITELN ('SORRY, YOU''RE OUT OF MONEY')
END.
```


$1=\frac{g^5+ab\sqrt{36^4+2b(3+a)}^3+2\sqrt{18}}{\frac{\sqrt{5}(a+2)^3+\left(a^5-3y^2+\frac{1}{4}b\right)^4-a\sqrt{6+b}}{3+\frac{12\sqrt{29}^{31}}{7a^{2(a+3)}}+\frac{197a(b+\sqrt{18})(3b+1)}{17\sqrt{b}^2\frac{3a}{3}\sqrt{23b}^3}}}$

# MATHGAME

## The Game.

Young students who dislike math exercises may find that practicing on the computer is a little more amusing than sitting at a desk with pencil, eraser, and mimeographed practice sheets.

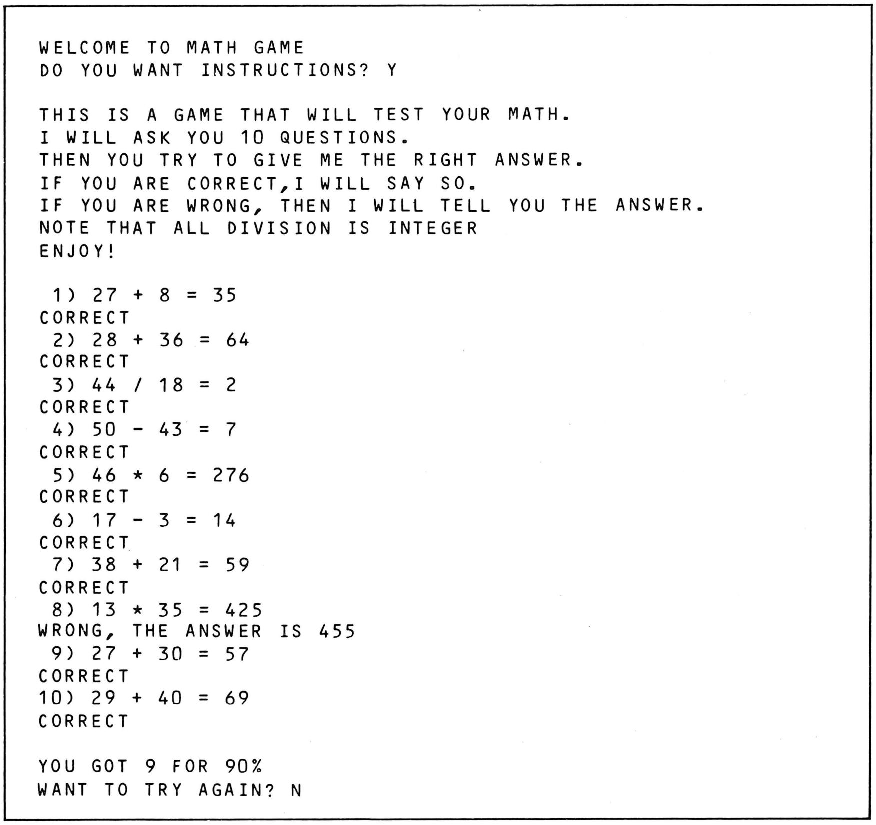
This program gives you ten simple arithmetic problems (addition, subtraction, multiplication, and division) to solve. When each problem is displayed, you are given the opportunity to type in your answer; there is no time limit. The program checks your answer and lets you know if you're right or wrong. If you're wrong, the program gives you the correct answer.

Any division problems that you are given should be answered with integral values only—without the "remainder." For example, the correct answer to

```
49 / 6 =
```

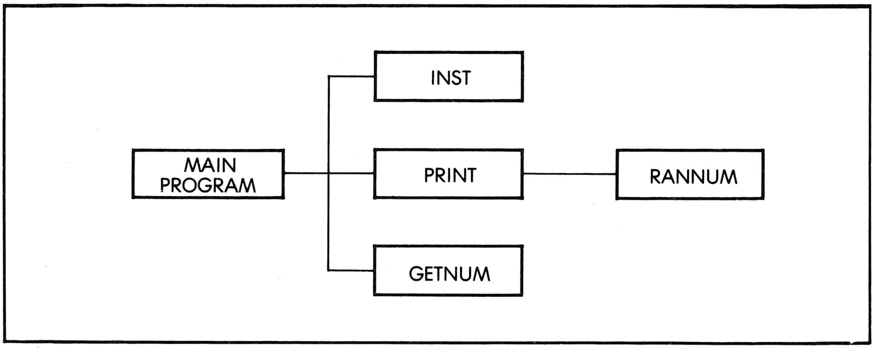
is 8.

After you have done the ten exercises, the program lets you know what percent you answered correctly. Then you may either quit (retreat to the kitchen for some graham crackers and milk) or try another round with a whole new set of exercises.



```
WELCOME TO MATH GAME
DO YOU WANT INSTRUCTIONS? Y

THIS IS A GAME THAT WILL TEST YOUR MATH.
I WILL ASK YOU 10 QUESTIONS.
THEN YOU TRY TO GIVE ME THE RIGHT ANSWER.
IF YOU ARE CORRECT,I WILL SAY SO.
IF YOU ARE WRONG, THEN I WILL TELL YOU THE ANSWER.
NOTE THAT ALL DIVISION IS INTEGER
ENJOY!

 1) 27 + 8 = 35
CORRECT
 2) 28 + 36 = 64
CORRECT
 3) 44 / 18 = 2
CORRECT
 4) 50 - 43 = 7
CORRECT
 5) 46 * 6 = 276
CORRECT
 6) 17 - 3 = 14
CORRECT
 7) 38 + 21 = 59
CORRECT
 8) 13 * 35 = 425
WRONG, THE ANSWER IS 455
 9) 27 + 30 = 57
CORRECT
10) 29 + 40 = 69
CORRECT

YOU GOT 9 FOR 90%
WANT TO TRY AGAIN? N
```

# The Program.



The main program begins by displaying a rather upbeat welcome message (WELCOME TO MATH GAME) and offering the player instructions. Procedure INST is called if the player types a Y.

The rest of the program is controlled by a **REPEAT/UNTIL** loop. An early call to RANDOMIZE anticipates the use of RANDOM in function PRINT. NUMRIGHT, the variable that holds the number of correct answers the player has given, is initialized to zero. Then a **FOR** loop, which repeats ten times, begins displaying the problems and accepting the answers.

Function PRINT is called in an assignment statement:

```
ANSWER := PRINT;
```

PRINT, however, does more than just return the answer to a problem; it also chooses and displays the problem. FIRSTNUM and SECNUM, the two numbers of the arithmetic problem, are determined from calls to function RANNUM. (RANNUM returns a random number from 1 to 50, using an algorithm we have already seen in several programs.) The operation (+, −, *, or /) is chosen randomly in a **CASE** statement:

```
CASE (1 + RANDOM MOD 4) OF . . .
```

The **CASE** "selector" is an expression that uses RANDOM to produce a number from 1 to 4. Each **CASE** label then corresponds to one of the four arithmetic operations. The answer to the problem is computed and assigned to the variable PRINT (to be returned to the main program), and the operation symbol is displayed. (These symbols were defined as the constants MULOP, DIVOP, SUBOP, and ADDOP in the **CONST** section of the main program.)

The division case could be easily rewritten to produce only division problems with no remainder:

```
2: BEGIN
     WRITE(FIRSTNUM * SECNUM, ' ');
     PRINT := FIRSTNUM;
     WRITE(DIVOP)
   END;
```

In this revision, the first number the player sees is actually the *product* of the two numbers returned by function RANNUM. FIRSTNUM then becomes the answer, and SECNUM becomes the divisor. For example, if FIRSTNUM is 22 and SECNUM is 12, the problem would be displayed as:

```
264 / 12 =
```

and the answer would be 22, the value of FIRSTNUM. Note that this revision also requires each **CASE** option to contain its own WRITE statement for the first number. For the other three operations this would simply be:

```
WRITE(FIRSTNUM, ' ');
```

Function GETNUM is called from the main program in an **IF** statement:

```
IF GETNUM <> ANSWER THEN . . .
```

GETNUM uses the implicitly defined buffer variable INPUT ^ to validate the first character typed in by the user:

```
IF NOT (INPUT ^ IN ['0..9','-']) THEN . . .
```

Thus the character must be a digit or a minus sign to be valid. For an illegal entry, an error message is displayed and another input is requested. Notice the rather cryptic statement:

```
GETNUM := GETNUM;
```

This is a recursive call to function GETNUM to receive the new input.

If GETNUM returns the correct answer to the main program, then the player is given a CORRECT message and NUMRIGHT is incremented by 1.

When all ten problems have been displayed and solved, the percentage of correct answers is computed and displayed:

```
WRITELN('YOU GOT ',NUMRIGHT,' FOR ',
NUMRIGHT*100 DIV TIMES,'%');
```

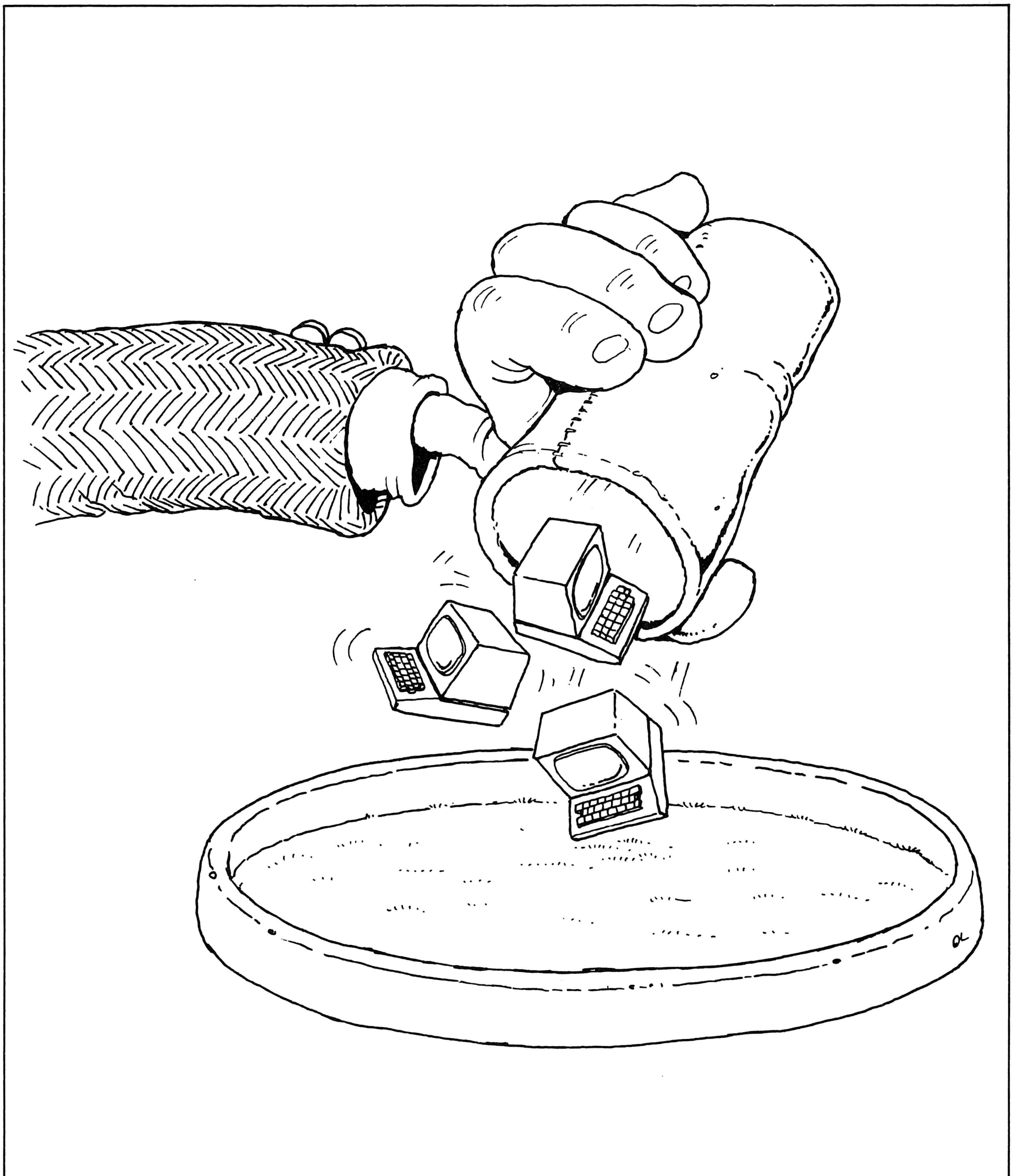
The player is then given the chance to jump back to the top of the **REPEAT/UNTIL** loop by typing a Y in response to:

```
WANT TO TRY AGAIN?
```

```
(*
 * MATHGAME
 * A SIMPLE GAME TO TEST YOUR ABILITIES
 * AT MATH
 *)
PROGRAM MATHGAME (INPUT,OUTPUT);
USES APPLESTUFF;                    (* FOR THE RANDOM FUNCTIONS *)
CONST
   TIMES     =   10;                (* NUMBER OF TIMES THE GAME IS PLAYED *)
   MAXNUM    =   50;                (* MAXIMUM NUMBER THAT WILL APPEAR *)
   MULOP     =   '*';               (* MULTIPLICATION OPERATOR *)
   DIVOP     =   '/';               (* DIVISION OPERATOR *)
   SUBOP     =   '-';               (* SUBTRACTION OPERATOR *)
   ADDOP     =   '+';               (* ADDITION OPERATOR *)
VAR
   NUMRIGHT   : 0..TIMES;           (* NUMBER OF CORRECT GUESSES *)
   CH         : CHAR;               (* CHARACTER FROM USER *)
   CURRENT    : 1..TIMES;           (* CURRENT PLAY NUMBER *)
   ANSWER     : INTEGER;            (* USER'S GUESS *)
   (*
    * RANNUM
    * RETURNS A RANDOM NUMBER BETWEEN 0
    * AND MAXNUM
    *)
   FUNCTION RANNUM:INTEGER;
   BEGIN
      RANNUM := RANDOM MOD (MAXNUM +1)
   END; (* RANNUM *)

   (*
    * PRINT
    * CHOOSES TWO NUMBERS, AN OPERATION,
    * AND ASKS FOR THE ANSWER. THIS
    * FUNCTION RETURNS THE CORRECT ANSWER
    *)
   FUNCTION PRINT:INTEGER;
   VAR
      FIRSTNUM, SECNUM : 0..MAXNUM;
      ANSWER : INTEGER;
   BEGIN
      FIRSTNUM := RANNUM;
      SECNUM := RANNUM;

      WRITE(FIRSTNUM,' ');
      CASE (1+RANDOM MOD 4) OF
         1 : BEGIN
                PRINT  := FIRSTNUM * SECNUM;
                WRITE(MULOP);
             END;

         2 : BEGIN
                IF SECNUM = 0 THEN
                   SECNUM := 1;
                PRINT  := FIRSTNUM DIV SECNUM;
                WRITE(DIVOP)
             END;
```

```
        3 : BEGIN
               PRINT   := FIRSTNUM - SECNUM;
               WRITE(SUBOP)
            END;

        4 : BEGIN
               PRINT   := FIRSTNUM + SECNUM;
               WRITE(ADDOP)
            END
   END; (* CASE *)

   WRITE(' ',SECNUM,' = ')
END; (* PRINT *)

(*
 * GETNUM
 * GET A NUMBER FROM THE PLAYER
 *)
FUNCTION GETNUM:INTEGER;
VAR
   CH : CHAR;                    (* INPUT FROM PLAYER *)
   ANSWER : INTEGER;             (* NUMBER PLAYER TYPES *)
BEGIN
   (* FIRST WAIT UNTIL A NUMBER IS TYPED *)
   WHILE INPUT^ = ' ' DO
      GET(INPUT);

   IF NOT (INPUT^ IN ['0'..'9','-']) THEN
   BEGIN
      READLN(CH);
      WRITELN('ILLEGAL INPUT, ONLY NUMBERS ALLOWED');
      WRITE('YOUR GUESS? ');
      GETNUM := GETNUM
   END
   ELSE
   BEGIN
      READLN(ANSWER);
      GETNUM := ANSWER
   END
END; (* GETNUM *)

(*
 * INST
 * PRINT OUT A LIST OF INSTRUCTIONS
 *)
PROCEDURE INST;
BEGIN
   WRITELN;
   WRITELN('THIS IS A GAME THAT WILL TEST YOUR MATH.');
   WRITELN('I WILL ASK YOU ',TIMES,' QUESTIONS.');
   WRITELN('THEN YOU TRY TO GIVE ME THE RIGHT ANSWER.');
   WRITELN('IF YOU ARE CORRECT,I WILL SAY SO.');
   WRITELN('IF YOU ARE WRONG, THEN I WILL TELL YOU THE ANSWER.');
   WRITELN('NOTE THAT ALL DIVISION IS INTEGER');
   WRITELN('ENJOY!');
   WRITELN
END; (* INST *)
```

```
BEGIN
   WRITELN('WELCOME TO MATH GAME');
   WRITE('DO YOU WANT INSTRUCTIONS? ');
   READLN(CH);
   IF CH = 'Y' THEN
      INST;

   REPEAT
      RANDOMIZE;
      NUMRIGHT := 0;
      FOR CURRENT := 1 TO TIMES DO

      BEGIN
         WRITE(CURRENT:2,') ');
         ANSWER := PRINT;
         IF GETNUM <> ANSWER THEN
            WRITELN('WRONG, THE ANSWER IS ',ANSWER)
         ELSE
         BEGIN
            WRITELN('CORRECT');
            NUMRIGHT := NUMRIGHT +1
         END
      END;
      WRITELN;
      WRITELN('YOU GOT ',NUMRIGHT,' FOR ',NUMRIGHT *100 DIV TIMES,'%');
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
   UNTIL CH <> 'Y';

END. (* MATHGAME *)
```


OL

# CHUCKALUCK

**The Game.** Here is a game of chance that uses three dice. The computer asks you to pick a number from 1 to 6 and to place a bet. Three dice are thrown and the roll is displayed for you. If your number matches one of the dice, you win your bet. For two matches you win twice your bet, and for three you win three times your bet. If there are no matches, on the other hand, you lose the amount of your bet.

For example, let's say you choose 3 as your number, and you bet $75. The dice are rolled. If the roll is

```
5 3 6
```

you win $75. For a luckier roll of

```
2 3 3
```

you win $150. If you're really lucky, and the roll turns up

```
3 3 3
```

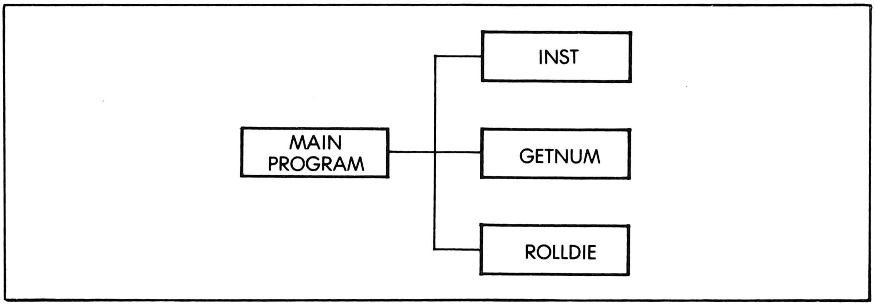
then you win $225.

You begin this game with $1000, and the game ends either when you go broke or when you call it quits by typing an 'N' in response to:

```
CARE TO TRY YOUR LUCK AGAIN?
```

```
WANT INSTRUCTIONS? Y
CHUCKALUCK

YOU CHOOSE A NUMBER BETWEEN 1 AND 6.
I ROLL THREE DICE, IF YOU MATCH 1, YOU
GET EVEN MONEY. IF YOU MATCH 2 DICE,
YOU GET DOUBLE, IF YOU MATCH ALL THREE,
YOU GET TRIPLE
YOU HAVE $1000
HOW LARGE A BET? 300
PICK A NUMBER 4
DIE1  DIE2  DIE3
   5     2     2
YOU LOST $300
CARE TO TRY YOUR LUCK AGAIN? Y
YOU HAVE $700
HOW LARGE A BET? 100
PICK A NUMBER 3
DIE1  DIE2  DIE3
   1     1     3
YOU WON $100
CARE TO TRY YOUR LUCK AGAIN? Y
YOU HAVE $800
HOW LARGE A BET? 200
PICK A NUMBER 2
DIE1  DIE2  DIE3
   1     1     4
YOU LOST $200
CARE TO TRY YOUR LUCK AGAIN? Y
YOU HAVE $600
HOW LARGE A BET? 200
PICK A NUMBER 5
DIE1  DIE2  DIE3
   3     5     1
YOU WON $200
CARE TO TRY YOUR LUCK AGAIN? N
YOU LEAVE WITH $800
```

# The Program.



As we will see, several details of this program are similar to features we have described in other games. The main program begins by giving the player $1000 (STARTMONEY) and calling procedure INST if the player indicates that instructions are needed. A **REPEAT/UNTIL** loop then takes control of the program, and continues until one of two conditions is met: the player runs out of money (MONEY < = 0) or the player wants to quit (CH <> Y).

After the player's present assets (MONEY) are displayed, the main program makes two calls to function GETNUM and assigns the returned values to BET and GUESS respectively. For each call to GETNUM, a string and two integer values are passed to the formal parameters QUESTION, MIN, and MAX. The string will be used as a prompt for the player to input a value (BET or GUESS), and the MIN and MAX values will be used to test the validity of the input value.

GETNUM performs two familiar validation tests on the input. First, it uses the GET procedure and the implicitly defined buffer variable INPUT ^ to insure a numeric input. Second, GETNUM is called *recursively* if the input value is outside of the range defined by MIN and MAX. (See GUESSIT for a more complete description of these Pascal features.)

Following the calls to GETNUM, the main program initializes MATCHED (the number of matches between the player's guess and the roll of the dice) to zero. A heading is then written. The calls to ROLLDIE are contained within the next **FOR** loop:

```
FOR EACHDIE := 1 TO NUMDIE DO
  IF ROLLDIE = GUESS THEN
    MATCHED := MATCHED + 1
```

This is a rather economical statement. It makes three calls to ROLLDIE, tests each returned value to see if it matches GUESS, and increments MATCHED if a match is found.

Function ROLLDIE is basically the same as other dice-rolling procedures we have seen: it combines calls to RANDOM with the **MOD** operation to produce the desired range of random numbers (1 to 6). The only additional feature of this function ROLLDIE is a WRITE statement that displays each roll of the die as it is produced.

Finally, a series of **IF/THEN/ELSE** statements in the main program test the value of MATCHED and adjust the value of MONEY accordingly. For a win, the product of BET and the payoff factor (defined in the **CONST** section at the beginning of the program) is added to MONEY. For example:

MONEY := MONEY + BET * PAYOFF2

For a loss, BET is subtracted from MONEY.

As long as there is MONEY to spend (MONEY > 0), the program keeps encouraging the player to try again.

```
(*
 * CHUCKALUCK
 * YOU CHOOSE A NUMBER BETWEEN 1 AND 6.
 * I ROLL THREE DICE, IF YOU MATCH 1, YOU
 * GET EVEN MONEY. IF YOU MATCH 2 DICE,
 * YOU GET DOUBLE, IF YOU MATCH ALL THREE,
 * YOU GET TRIPLE
 *)
PROGRAM CHUCKALUCK(INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   STARTMONEY   =  1000;       (* AMOUNT OF MONEY YOU START WITH *)

   MAXDIE       =     6;       (* MAXIMUM VALUE ON A DIE *)
   MINDIE       =     1;       (* MINIMUM    "    "  "   "  *)

   NUMDIE       =     3;       (* THE NUMBER OF DICE *)

   PAYOFF1      =     1;       (* PAY OFF EVEN MONEY ON 1 MATCH *)
   PAYOFF2      =     2;       (* PAY OFF DOUBLE ON TWO *)
   PAYOFF3      =     3;       (* PAY OFF TRIPLE ON THREE *)
VAR
   MONEY   : INTEGER;          (* THE AMOUNT OF MONEY YOU HAVE *)
   BET     : INTEGER;          (* AMOUNT OF THE BET *)
   GUESS   : MINDIE..MAXDIE;   (* THE DIE YOU GUESS *)
   CH      : CHAR;             (* USED FOR INPUT *)
   EACHDIE: 1..NUMDIE;         (* INDEX USED TO ROLL DIE *)
   MATCHED: 0..NUMDIE;         (* NUMBER OF MATCHED DICE *)
  (*
   * INST
   * PRINTS OUT INSTRUCTIONS
   *)
  PROCEDURE INST;
  BEGIN

     WRITELN('CHUCKALUCK');
     WRITELN;
     WRITELN('YOU CHOOSE A NUMBER BETWEEN 1 AND 6.');
     WRITELN('I ROLL THREE DICE, IF YOU MATCH 1, YOU');
     WRITELN('GET EVEN MONEY. IF YOU MATCH 2 DICE,');
     WRITELN('YOU GET DOUBLE, IF YOU MATCH ALL THREE,');
     WRITELN('YOU GET TRIPLE');
     WRITELN
  END; (* INST *)
```

```
   (*
    * GETNUM
    * ASK A QUESTION FROM A USER,
    * AND GET A NUMBER BETWEEN MIN AND
    * MAX (WITH ERROR CHECKING)
    *)
   FUNCTION GETNUM(QUESTION:STRING;MIN,MAX:INTEGER):INTEGER;
   VAR
      INTER : INTEGER;        (* INTERMEDIATE RESULT *)
   BEGIN
      WRITE(QUESTION);
      GET(INPUT);
      RANDOMIZE;
      WHILE NOT (INPUT^ IN ['1'..'9']) DO
      BEGIN
         READLN(INPUT);
         WRITELN('SORRY, ONLY NUMBERS ARE ALLOWED');
         WRITE(QUESTION);
         GET(INPUT)


      END;
      READLN(INTER);
      IF (INTER < MIN) OR (INTER > MAX) THEN
      BEGIN
         WRITELN('ONLY NUMBERS BETWEEN ',MIN,' AND ',MAX,' ARE ALLOWED');
         GETNUM := GETNUM(QUESTION,MIN,MAX)
      END
      ELSE
         GETNUM := INTER
   END; (* GETNUM *)


   (*
    * ROLLDIE
    * ROLL A DIE, AND PRINT OUT ITS
    * VALUE.
    *)
   FUNCTION ROLLDIE:INTEGER;
   VAR
      DIEVALUE : MINDIE..MAXDIE;
   BEGIN
      DIEVALUE := MINDIE + RANDOM MOD MAXDIE;

      WRITE(DIEVALUE:4,'  ');
      ROLLDIE := DIEVALUE
   END; (* ROLLDIE *)


BEGIN
   MONEY := STARTMONEY;

   WRITE('WANT INSTRUCTIONS? ');
   READLN(CH);
   IF CH = 'Y' THEN
      INST;
   REPEAT
      WRITELN;
      WRITELN('YOU HAVE $',MONEY);
      BET := GETNUM('HOW LARGE A BET? ',1,MONEY);
      GUESS := GETNUM('PICK A NUMBER ',MINDIE,MAXDIE);

      MATCHED := 0;
      WRITELN('DIE1  DIE2  DIE3');
      FOR EACHDIE := 1 TO NUMDIE DO
         IF ROLLDIE = GUESS THEN
            MATCHED := MATCHED +1;

      WRITELN;

      IF MATCHED = 1 THEN
      BEGIN
         WRITELN('YOU WON $',BET*PAYOFF1);
         MONEY := MONEY+BET*PAYOFF1
      END
      ELSE
      IF MATCHED = 2 THEN
      BEGIN
         WRITELN('YOU WON $',BET*PAYOFF2);
         MONEY := MONEY + BET*PAYOFF2
      END

      ELSE
      IF MATCHED = 3 THEN
      BEGIN
         WRITELN('YOU WON $',BET*PAYOFF3);
         MONEY := MONEY + BET*PAYOFF3
      END
      ELSE
      BEGIN
         WRITELN('YOU LOST $',BET);
         MONEY := MONEY - BET
      END;

      IF MONEY > 0 THEN
      BEGIN
         WRITE('CARE TO TRY YOUR LUCK AGAIN? ');
         READLN(CH)
      END

   UNTIL (MONEY <= 0) OR (CH <> 'Y');

   WRITELN('YOU LEAVE WITH $',MONEY)

END. (* CHUCKALUCK *)
```

# BOGGLE

**The Game.** This is a Pascal version of a popular number guessing game. The program chooses a four-digit number at random, and you must develop a strategy for guessing the number in as few turns as possible.

Each time you type in a guess, the program will give you two pieces of information:

- the number of digits of your guess that are *both* correct *and* in the right place (these are your "white numbers")
- the number of digits of your guess that are correct, but not in the right place (these are your "black numbers")

For example, if you get a message like this:

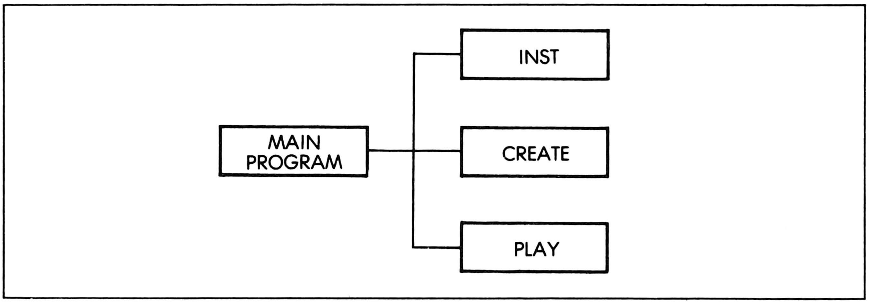
```
WHITE = 1
BLACK = 2
```

you know that three of the digits in your guess are correct, but only one of them is in the right place.

When you guess the number (i.e., all four digits correct and in place) the program tells you how many guesses you took and gives you a chance to guess a new number.

```
DO YOU WANT INSTRUCTIONS? Y
BOGGLE
A VERSION OF MASTER MIND.
I GUESS A NUMBER, THEN YOU ATTEMPT
TO FIND OUT WHAT THE NUMBER IS. WHEN
YOU MAKE A GUESS, I WILL TELL YOU HOW
MANY DIGITS WERE CORRECT, AND IN THEIR
CORRECT LOCATION (WHITE NUMBERS). I
WILL ALSO TELL YOU HOW MANY DIGITS WERE
CORRECT, BUT IN THE WRONG PLACE.
FOR EXAMPLE, IF THE NUMBER IS 1234, AND
YOU GUESS 1467, THEN YOU WOULD HAVE
1 WHITE (THE 1),AND 1 BLACK NUMBER
(THE 4).
YOUR GUESS? 1234
WHITE = 1
BLACK = 1
YOUR GUESS? 5678
WHITE = 1
BLACK = 1
YOUR GUESS? 1256
WHITE = 0
BLACK = 2
YOUR GUESS? 5623
WHITE = 1
BLACK = 2
YOUR GUESS? 7623
WHITE = 0
BLACK = 3
YOUR GUESS? 5723
WHITE = 2
BLACK = 2
YOUR GUESS? 5732
WHITE = 4
BLACK = 0
CORRECT
IT TOOK YOU 7 GUESSES
CARE TO TRY AGAIN? N
```

# The Program.



Procedure INST, which contains a description of this deceptively simple-sounding game,[1] is called once at the beginning of the main program if the player wants instructions. Next, a **REPEAT/UNTIL** loop controls the rounds of the game. Since procedure CREATE uses the RANDOM function, a call is made to RANDOMIZE before control is passed to CREATE.

Procedure CREATE produces the four-digit secret number and assigns the values, in character form, to the array NUMBER. Each digit of the number is chosen randomly, but CREATE builds the set USED to make sure that all four digits will be different. USED is initially a null set:

```
USED := [ ];
```

As each digit, TEMP, is chosen, TEMP is added to the set:

```
USED := USED + [TEMP]
```

Each new value of TEMP is chosen within a **REPEAT** loop, which assures that no digit will appear twice in NUMBER:

```
REPEAT
   TEMP := CHR(ORD('0') + RANDOM MOD 10)
UNTIL NOT (TEMP IN USED);
NUMBER[I] := TEMP;
```

RANDOM **MOD** 10 produces a digit from 0 to 9. To transform this digit to a character, we first add ORD('0') to compute the correct character code; then the CHR function returns the character value. We will see that the array NUMBER can be compared easily to the string USER, which contains the player's guess.

Back in the main program, NUMWHITE (the number of correct digits) and NUMGUESS (the number of guesses the player has made) are initialized to zero. Then a **WHILE** loop controls the calls to procedure PLAY and the evaluation of each guess. The **WHILE** loop continues until NUMWHITE equals NUMDIGITS—that is, until all four digits are correct and in the right place.

Procedure PLAY reads the player's input, makes sure the input represents a valid guess, then counts up the number of "white numbers" and "black numbers" in the guess. PLAY uses the BOOLEAN variable NOTBAD and the set USED to validate the player's input. The input must be the correct length (four digits long):

---

[1] See *Inside BASIC Games*, by Richard Mateosian (SYBEX, 1981), for a complete discussion of the BASIC version of this game.

```
NOTBAD := LENGTH(USER) = NUMDIGITS;
```

And the input string must consist of four different digits from '0' to '9':

```
NOTBAD := (USER[I] IN ['0'..'9']) AND
                (NOT (USER[I] IN USED));
```

Note that USED is initialized to the null set at the beginning of PLAY.

If the input string USER is invalid (**IF NOT** NOTBAD) then an error message is displayed, and PLAY is called recursively. Otherwise, two **FOR** loops are used to compare the elements of NUMBER and USER.

The first **FOR** loop is the simpler of the two. It tests for "white numbers":

```
FOR I := 1 TO NUMDIGITS DO
  IF NUMBER[I] = USER[I] THEN
    NUMWHITE := NUMWHITE + 1;
```

The corresponding elements of NUMBER and USER are compared. If a pair of elements are the same, NUMWHITE is incremented by 1.

Next, in a pair of nested **FOR** loops, each element of USER is compared to all of the elements of NUMBER to find the "black numbers." Each comparison is subjected to two tests:

```
IF (USER[I] = NUMBER[J]) AND (I <> J)
```

That is, if a pair of elements are equal (USER[I] = NUMBER[J]), and the match does not represent a "white number" (I <> J) then NUMBLACK is incremented by 1.

Returning once again to the main program, we see that NUMWHITE and NUMBLACK are displayed for the player after each guess. When the entire number is correct, the program tells the player how many guesses were needed, and offers another round.

```
PROGRAM BOGGLE (INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   NUMDIGITS    =  4;           (* THE NUMBER OF DIGITS IN THE NUMBER *)
VAR
   CH      : CHAR;              (* USED TO ANSWER QUESTIONS *)
   NUMBER : ARRAY[1..NUMDIGITS]OF CHAR; (* THE ACTUAL NUMBER *)
   NUMBLACK,                    (* NUMBER OF CORRECT DIGITS *)
   NUMWHITE: 0..NUMDIGITS;      (* NUMBER OF CORRECT AND ORDERED DIGITS *)
   NUMGUESS: INTEGER;           (* NUMBER OF GUESSES TAKEN *)

   (*
    * CREATE
    * CREATE A RANDOM NUMBER
    *)
   PROCEDURE CREATE;
   TYPE
     RANGE = '0'..'9';

   VAR
     I : 1..NUMDIGITS;
     TEMP : RANGE;
     USED : SET OF RANGE;

   BEGIN
     USED := [];
     FOR I := 1 TO NUMDIGITS DO
       BEGIN
         REPEAT
           TEMP := CHR(ORD('0') + RANDOM MOD 10)
         UNTIL NOT (TEMP IN USED);
         NUMBER[I] := TEMP;
         USED := USED + [TEMP]
       END
   END;    (* CREATE *)


(*
 * PLAY
 * GETS A NUMBER FROM THE USER,
 * THEN FIGURES OUT HOW MANY WHITE,
 * AND BLACK NUMBERS HE (SHE) HAS
 *)
PROCEDURE PLAY;
CONST
   INDEXLENGTH   = 5;         (* LENGTH OF THE INDICES *)
VAR
   USER  : STRING;            (* INPUT FROM USER *)
   NOTBAD: BOOLEAN;           (* USED TO TEST FOR GARBAGE *)
   I,J   : 1..INDEXLENGTH;    (* INDEX VARIABLES *)
   USED  : SET OF '0'..'9'; (* WHETHER THE DIGIT HAS BEEN USED *)

BEGIN
   USED := [];
   NUMWHITE := 0;
   NUMBLACK := 0;
   WRITE('YOUR GUESS? ');
   READLN(USER);
   NOTBAD := LENGTH(USER) = NUMDIGITS;

   I := 1;
   WHILE NOTBAD AND (I <= NUMDIGITS) DO
   BEGIN
      NOTBAD := (USER[I] IN ['0'..'9'])
                AND (NOT (USER[I] IN USED));
      USED := USED + [USER[I]];
      I := I + 1
   END;

   IF NOT NOTBAD THEN
   BEGIN
      WRITELN('ILLEGAL INPUT, TRY AGAIN');
      PLAY
   END
   ELSE
   BEGIN
      FOR I := 1 TO NUMDIGITS DO
         IF NUMBER[I] = USER[I] THEN
            NUMWHITE := NUMWHITE +1;

      FOR I := 1 TO NUMDIGITS DO
        FOR J:= 1 TO NUMDIGITS DO
          IF (USER[I] = NUMBER[J]) AND (I<>J) THEN
            NUMBLACK := NUMBLACK + 1

   END
END; (* PLAY *)


(*
 * INST
 * PRINT OUT THE INSTRUCTIONS
 *)
PROCEDURE INST;
BEGIN
   WRITELN('BOGGLE');
   WRITELN;
   WRITELN('A VERSION OF MASTER MIND.');
   WRITELN('I GUESS A NUMBER, THEN YOU ATTEMPT');
   WRITELN('TO FIND OUT WHAT THE NUMBER IS. WHEN');
   WRITELN('YOU MAKE A GUESS, I WILL TELL YOU HOW');
   WRITELN('MANY DIGITS WERE CORRECT, AND IN THEIR');
   WRITELN('CORRECT LOCATION (WHITE NUMBERS). I');
   WRITELN('WILL ALSO TELL YOU HOW MANY DIGITS WERE ');
   WRITELN('CORRECT, BUT IN THE WRONG PLACE.');
   WRITELN('FOR EXAMPLE, IF THE NUMBER IS 1234, AND');
   WRITELN('YOU GUESS 1467, THEN YOU WOULD HAVE');
   WRITELN('1 WHITE (THE 1),AND 1 BLACK NUMBER');
   WRITELN('(THE 4).')
END; (* INST *)
```

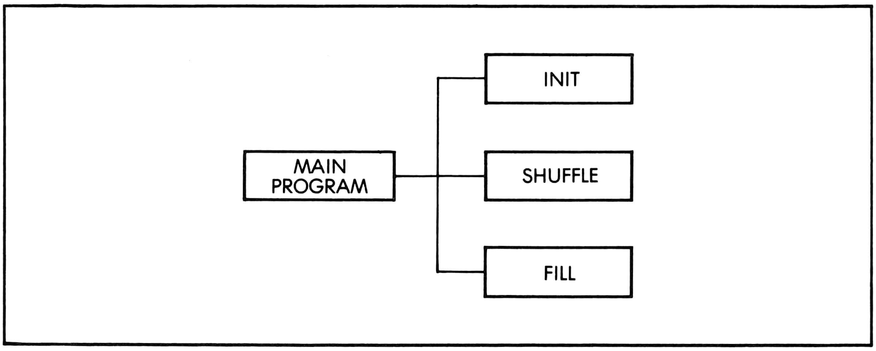
```
BEGIN
   WRITE('DO YOU WANT INSTRUCTIONS? ');
   READLN(CH);
   IF CH = 'Y' THEN
      INST;

   REPEAT
      RANDOMIZE;
      CREATE;
      NUMWHITE := 0;
      NUMGUESS := 0;
      WHILE NUMWHITE <> NUMDIGITS DO
      BEGIN
         PLAY;
         WRITELN('WHITE = ',NUMWHITE);
         WRITELN('BLACK = ',NUMBLACK);
         NUMGUESS := NUMGUESS +1
      END;
      WRITELN;
      WRITELN('CORRECT');
      WRITELN('IT TOOK YOU ',NUMGUESS,' GUESSES');
      WRITELN;
      WRITE('CARE TO TRY AGAIN? ');
      READLN(CH)
   UNTIL CH <> 'Y'
END. (* BOGGLE *)
```

# TWINKLE

**The Game.** Twinkle is an amusing little exercise in computer graphics. It turns the darkness of your screen into light by printing asterisks in a random pattern until the screen is full. It then reverses the process, removing the asterisks one by one until the screen is dark again.

You need provide no input at all. Just run the program and watch what happens.

## The Program.



The primary data type of program TWINKLE is POS, a record with two fields: XLOC and YLOC. The variable SCREEN, an array of POS, will be used to assign x and y coordinates to all 960 (i.e., 40 by 24) character positions of the screen. (Notice that SCRSIZE, XMAX, and YMAX are all defined as constants at the beginning of the program heading; thus they can be easily changed for a different screen size.)

The main program contains five procedure calls: one to INIT (to clear the screen and initialize SCREEN) and two each to SHUFFLE and FILL (first to fill the screen with asterisks, then to take them away again).

INIT clears the screen via a **FOR** loop that performs a WRITELN statement 24 times. (In other words, it writes a blank line on all 24 lines of the screen.) The variable SCREEN is initialized inside two levels of **FOR** loops. The inner loop increments the *y* coordinate, and the outer loop increments the x coordinate. Three statements are performed inside the loops:

```
SCREEN[SCRINDEX].XLOC := X;
SCREEN[SCRINDEX].YLOC := Y;
SCRINDEX := SCRINDEX + 1
```

The first two statements assign values to the fields of each element of SCREEN. The third statement increments the index into SCREEN. Thus, for example, the position in the lower right-hand corner of the screen will be represented by the 960th element of SCREEN, and will have the coordinates (39,23):

```
SCREEN[960].XLOC := 39;
SCREEN[960].YLOC := 23;
```

Procedure SHUFFLE randomizes the order of the elements of SCREEN so that SCREEN[1] to SCREEN[960] represent an unpredictable order of (x,y) coordinates. (The SHUFFLE algorithm is used elsewhere in this book to shuffle cards; see Baccarat and Cribbage.) The "shuffling" occurs inside a **FOR** loop that increments the index I through all the elements of SCREEN. RND is assigned a random number from 1 to 960:

```
RND := (RANDOM MOD SCRSIZE) + 1;
```

Then the value of each element of screen is exchanged, randomly, with the value of another element of SCREEN. (Note the use of the temporary storage variable, TMP.)

Procedure FILL is the simplest of the three procedures. It receives a single character from the main program:

```
FILL('*');
```

or

```
FILL(' ')
```

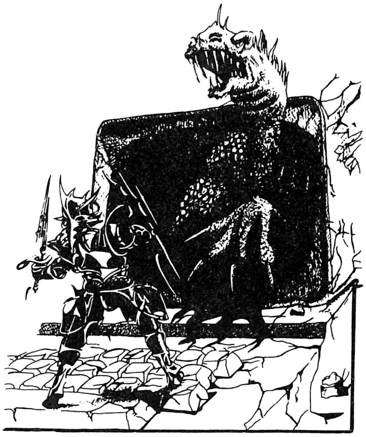
and assigns that character to the formal parameter CH. A **FOR** loop increments the index I from 1 to 960. Using the UCSD procedure GOTOXY, the cursor is moved to each of the screen coordinates represented by the fields of SCREEN, and the character CH is written there:

```
GOTOXY(SCREEN[I].XLOC,SCREEN[I].YLOC);
WRITE(CH)
```

```
 (*
 * TWINKLE
 * RANDOMLY FILL THE SCREEN WITH
 * '*', AND THEN REMOVE THEM RANDOMLY
 *)
PROGRAM TWINKLE(OUTPUT);
USES APPLESTUFF;
CONST
   SCRSIZE = 960;       (* NUMBER OF POINTS ON THE SCREEN *)

   XMAX    = 39;        (* MAXIMUM X LOCATION (0 -> 39) *)
   YMAX    = 23;        (* MAXIMUM Y LOCATION *)
TYPE
   POS = RECORD
      XLOC : 0..XMAX;   (* X LOCATION *)
      YLOC : 0..YMAX
   END;

   INDEX = 1..SCRSIZE;  (* SCREEN INDEX *)

VAR
   SCREEN : ARRAY[INDEX] OF POS;        (* SCREEN POSITIONS *)

   (*
    * INIT
    * DO A PRIMITIVE CLEAR SCREEN,
    * THEN INIT THE SCREEN ARRAY
    *)
```

```
PROCEDURE INIT;
VAR
   SCRINDEX :INTEGER;(* INDEX INTO SCREEN ARRAY *)
   X : 0..XMAX;
   Y : 0..YMAX;
BEGIN
   (* DO A SIMPLE CLEAR SCREEN *)
   FOR Y := 0 TO YMAX DO
      WRITELN;

   RANDOMIZE;

   SCRINDEX := 1;
   FOR X := 0 TO XMAX DO
      FOR Y := 0 TO YMAX DO
      BEGIN
         SCREEN[SCRINDEX].XLOC := X;
         SCREEN[SCRINDEX].YLOC := Y;
         SCRINDEX := SCRINDEX + 1
      END
END; (* INIT *)

(*
 * SHUFFLE
 * SHUFFLE THE SCREEN INDEX
 *)
PROCEDURE SHUFFLE;
VAR
   I : INDEX;        (* SCREEN INDEX *)
   TMP : POS;        (* TEMP PLACE TO STORE VALUE *)
   RND : INDEX;      (* REPLACE I'TH MEMBER BY THIS ONE *)
BEGIN
   FOR I := 1 TO SCRSIZE DO
   BEGIN
      RND := (RANDOM MOD SCRSIZE) + 1;
      TMP := SCREEN[RND];
      SCREEN[RND] := SCREEN[I];
      SCREEN[I] := TMP
   END
END; (* SHUFFLE *)

(*
 * FILL
 * FILL UP THE SCREEN WITH A CHARACTER
 *)
PROCEDURE FILL(CH : CHAR);
VAR
   I : INDEX;
BEGIN
   FOR I := 1 TO SCRSIZE DO
   BEGIN
      GOTOXY(SCREEN[I].XLOC,SCREEN[I].YLOC);
      WRITE(CH)
   END
END; (* FILL *)
```

```
   (* MAIN *)
BEGIN
   INIT;
   SHUFFLE;
   FILL ('*');
   SHUFFLE;
   FILL(' ')
END. (* TWINKLE *)
```



# BLOCK

```
P
PPPPPP
P    P
P    P
PPPPPP
P
P
P

A
AAAAAA
A    A
A    A
AAAAAA
A    A
A    A
A    A

S
SSSSSS
S
S
SSSSSS
     S
     S
SSSSSS

C
CCCCCC
C
C
C
C
C
CCCCCC

A
AAAAAA
A    A
A    A
AAAAAA
A    A
A    A
A    A

L
L
L
L
L
L
L
LLLLLL
```

## The Game.

Here the computer tries its hand at large-scale calligraphy. A somewhat modest attempt, perhaps, but everyone has to start somewhere.

The program waits for your input. Type in as many letters as you wish, and the computer will draw them for you. Terminate the program by pressing the RETURN key.

## The Program.

In spite of its length, this program is extremely simple. The main program reads the character to be printed, and stores the input in the global variable CH. It then calls DRAW.

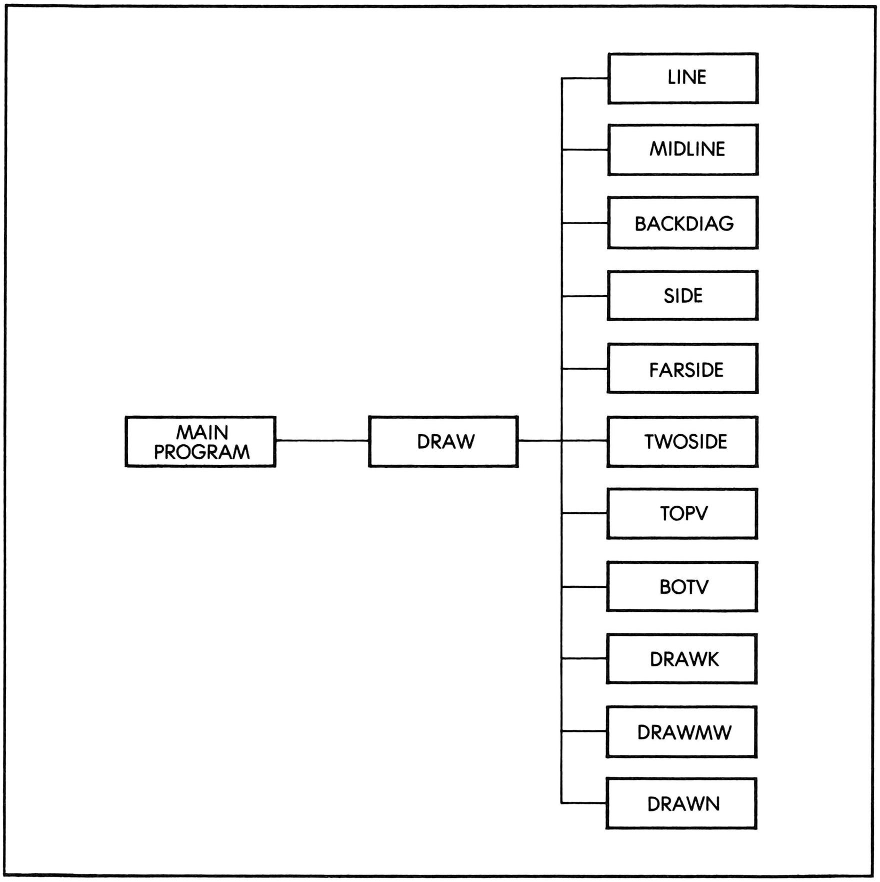
Procedure DRAW is made up of one long **CASE** statement that provides a series of procedure calls for each letter. For the most part, there are seven procedure calls for a letter; the block letters are seven characters high and six characters wide.

Procedures LINE, MIDLINE, BACKDIAG, SIDE, FARSIDE, TWOSIDE, TOPV, and BOTV are the generalized procedures called by DRAW. Most of them write only one or two characters for each call (though BACKDIAG, TOPV, and BOTV write more complicated patterns). The character that is used to build the block letter is the letter itself (i.e., the value of the global variable CH).

Several letters require more elaborate algorithms than the rest, and thus are drawn by special procedures of their own. These include K (procedure DRAWK), N (procedure DRAWN), and M and W (procedure DRAWMW). Procedure DRAWMW is the most difficult of these procedures. It is called with a parameter of type BOOLEAN, passed to the formal parameter ISM in DRAWMN. The value of ISM is thus TRUE for M and FALSE for W:

```
        'M' : DRAWMW(TRUE);
          . . .
        'W' : DRAWMW(FALSE);
```

The action of the procedure then depends on the values of three variables (BY, I, and STOP) that are initialized either for the M or the W.


MAIN PROGRAM
DRAW
LINE
MIDLINE
BACKDIAG
SIDE
FARSIDE
TWOSIDE
TOPV
BOTV
DRAWK
DRAWMW
DRAWN

```
(*
 * BLOCK
 * PRINT OUT BLOCK LETTERS
 *)
PROGRAM BLOCK(INPUT,OUTPUT);
CONST
   LENGTH       =  6;            (* LENGTH OF A LETTER *)
   LENGTH1      =  7;
VAR
   CH   : CHAR;                  (* CHARACTER TO ENLARGE *)

   (*
    * LINE
    * DRAW A LINE
    *)
   PROCEDURE LINE;
   VAR
      INDEX : 1..LENGTH;
   BEGIN
      FOR INDEX := 1 TO LENGTH DO
         WRITE(CH);
      WRITELN
   END; (* LINE *)

   (*
    * MIDLINE
    * PRINT OUT A LINE HALF WAY ACROSS
    * THE SCREEN
    *)
   PROCEDURE MIDLINE;
   VAR
      I : 1..LENGTH;
   BEGIN
      FOR I := 1 TO LENGTH DIV 2 DO
         WRITE(' ');
      WRITELN(CH)
   END; (* MIDLINE *)

   (*
    * BACKDIAG
    * DRAW A BACKDIAGONAL
    *)
   PROCEDURE BACKDIAG;
   VAR
      J,I : 0..LENGTH;
   BEGIN
      FOR J := 1 TO LENGTH  DO
      BEGIN
         FOR I := 1 TO LENGTH -J  DO
            WRITE(' ');
         WRITELN(CH)
      END
   END; (* BACKDIAG *)
```

```
(*
 * SIDE
 * DRAW A SINGLE SIDE
 *)
PROCEDURE SIDE;
BEGIN
   WRITELN(CH)
END; (* SIDE*)

(*
 * FARSIDE
 * DRAW A LINE ON THE FAR SIDE
 *)
PROCEDURE FARSIDE;
VAR
   I : 1..LENGTH;
BEGIN
   FOR I := 1 TO LENGTH -1 DO
      WRITE(' ');
   WRITELN(CH)
END; (* FAR SIDE *)


(*
 * TWOSIDE
 * DRAW A '*   *'
 *)
PROCEDURE TWOSIDE;
VAR
   INDEX : 1..LENGTH;
BEGIN
   WRITE(CH);
   FOR INDEX := 2 TO LENGTH -1 DO
      WRITE(' ');
   SIDE
END; (* TWOSIDE *)



(*
 * TOPV
 * DRAW A  V ON THE TOP OF THE LETTER
 * USED FOR X,Y, AND V
 *)
PROCEDURE TOPV;
VAR
   I,J : 0..LENGTH;
   D : 0..LENGTH1;
BEGIN
   FOR I := 1 TO ROUND(LENGTH / 2) DO
   BEGIN
      FOR J := 1 TO I -1 DO
         WRITE(' ');
      WRITE(CH);
      IF (CH = 'V') THEN D := I + 1
      ELSE D := I;
      FOR J := I TO LENGTH - D DO
         WRITE(' ');
      WRITELN(CH)
   END
END; (* TOPV *)

(*
 * BOTV
 * DRAW AN UPSIDE DOWN V, ON THE
 * BOTTOM OF THE SCREEN (USED IN X)
 *)
PROCEDURE BOTV;
VAR
   I,J : 0..LENGTH;
BEGIN
   FOR I := ROUND(LENGTH /2) DOWNTO 1 DO
   BEGIN
      FOR J := I-1 DOWNTO 1 DO
          WRITE(' ');
       WRITE(CH);
       FOR J := LENGTH -I DOWNTO I DO
          WRITE(' ');
       WRITELN(CH)
    END
 END; (* BOTV *)

 (*
  * WHAT FOLLOWS HERE ARE THE SPECIAL CASE
  * LETTERS. THEY MUST BE DRAWN BY
  * THEMSELVES.
  *)

 (*
  * DRAWK
  * DRAW A K
  *)
 PROCEDURE DRAWK;
 VAR
    I, J : 0..LENGTH;
 BEGIN
    FOR I := 1 TO TRUNC(LENGTH / 2) DO
    BEGIN
       WRITE(CH);
       FOR J := I -1 TO (LENGTH DIV 2) -1 DO
         WRITE(' ');
       WRITELN(CH)
    END;
```

```
     FOR I := ROUND(LENGTH / 2) DOWNTO 1 DO
     BEGIN
        WRITE(CH);
        FOR J := I-1 TO (LENGTH DIV 2) -1 DO
           WRITE(' ');
        WRITELN(CH)
     END
  END; (* DRAWK *)
  (*
   * DRAWMW
   * DRAW AN M OR W
   *)
  PROCEDURE DRAWMW (ISM : BOOLEAN);
  VAR
     I,J : 0..LENGTH;
     STOP : 0..LENGTH;
     BY : -1..1;
  BEGIN
     IF ISM THEN
     BEGIN
        BY := 1;
        I := 1;
        STOP := LENGTH DIV 2
     END
     ELSE
     BEGIN
        BY := -1;
        I := LENGTH DIV 2;
        STOP := 0
     END;
     TWOSIDE;
     WHILE I <> STOP DO
     BEGIN
        WRITE(CH);
        FOR J := 2 TO I DO
           WRITE(' ');
        WRITE(CH);
        IF I <> (LENGTH DIV 2) THEN
        BEGIN
           J := ABS(I + I + 2 - LENGTH);
           WHILE J > 0 DO
           BEGIN
              WRITE(' ');
              J := J - 1
           END;
           WRITE(CH);
           FOR J := 2 TO I DO
              WRITE(' ');
        END
        ELSE
           FOR J := 3 TO I DO
              WRITE(' ');;
        WRITELN(CH);
        I := I + BY
     END;
     TWOSIDE;
```

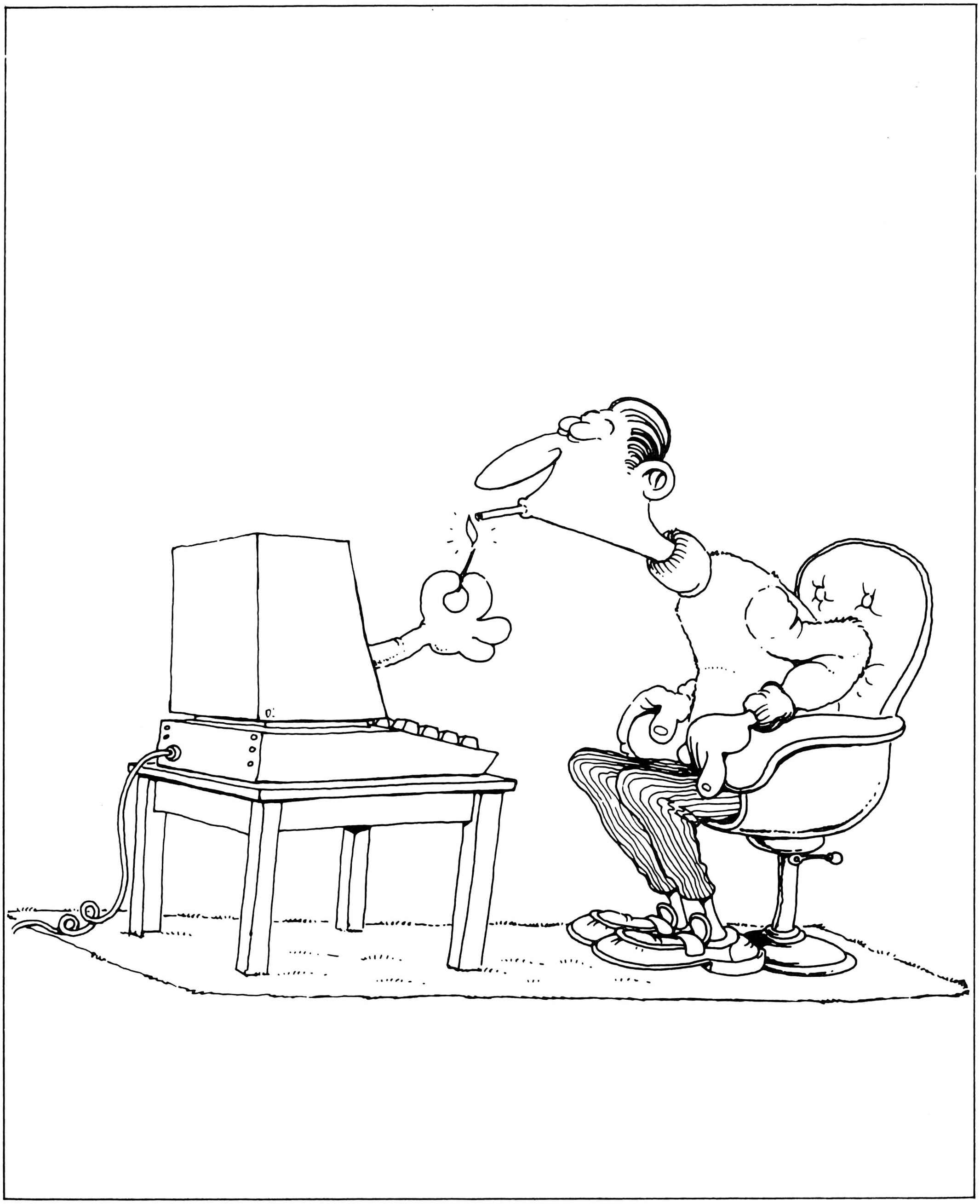
```
END; (* DRAWMW *)

(*
 * DRAWN
 * DRAW AN N
 *)
PROCEDURE DRAWN;
VAR
   I,J : 0..LENGTH;
BEGIN;
   FOR I := 1 TO LENGTH DO
   BEGIN
      WRITE(CH);
      FOR J := 2 TO LENGTH -1 DO
      BEGIN
         IF J = I THEN
            WRITE(CH)
         ELSE
            WRITE(' ')
      END;


      WRITELN(CH)
   END
END; (* DRAWN *)

(*
 * DRAW
 * CASE STATEMENT WHICH CALLS THE
 * NEEDED ROUTINES FOR EACH CHARACTER
 *)
PROCEDURE DRAW;
BEGIN
   WRITELN;
   IF CH IN ['A'..'Z'] THEN
   BEGIN
      CASE CH OF

         'A' : BEGIN
                  LINE;
                  TWOSIDE;
                  TWOSIDE;
                  LINE;
                  TWOSIDE;
                  TWOSIDE;
                  TWOSIDE
               END;
         'B' : BEGIN
                  LINE;
                  TWOSIDE;
                  TWOSIDE;
                  LINE;
                  TWOSIDE;
                  TWOSIDE;
                  LINE
               END;
```

```
        'C' : BEGIN
                LINE;
                SIDE;
                SIDE;
                SIDE;
                SIDE;
                SIDE;
                LINE
              END;
        'D' : BEGIN
                LINE;
                TWOSIDE;
                TWOSIDE;
                TWOSIDE;
                TWOSIDE;
                TWOSIDE;
                LINE
              END;
        'E' : BEGIN
                LINE;
                SIDE;
                SIDE;
                LINE;
                SIDE;
                SIDE;
                LINE
              END;
        'F' : BEGIN
                LINE;
                SIDE;
                SIDE;
                LINE;
                SIDE;
                SIDE;
                SIDE
              END;
        'G' : BEGIN
                LINE;
                SIDE;
                SIDE;
                TWOSIDE;
                TWOSIDE;
                TWOSIDE;
                LINE
              END;
        'H' : BEGIN
                TWOSIDE;
                TWOSIDE;
                TWOSIDE;
                LINE;
                TWOSIDE;
                TWOSIDE;
                TWOSIDE
              END;
```

```
'I' : BEGIN
          LINE;
          MIDLINE;
          MIDLINE;
          MIDLINE;
          MIDLINE;
          MIDLINE;
          LINE
      END;
'J' : BEGIN
          FARSIDE;
          FARSIDE;
          FARSIDE;
          FARSIDE;
          FARSIDE;
          FARSIDE;
          LINE
      END;

'K' : DRAWK;

'L' : BEGIN
          SIDE;
          SIDE;
          SIDE;
          SIDE;
          SIDE;
          SIDE;
          LINE
      END;

'M' : DRAWMW(TRUE);

'N' : DRAWN;

'O' : BEGIN
          LINE;
          TWOSIDE;
          TWOSIDE;
          TWOSIDE;
          TWOSIDE;
          TWOSIDE;
          LINE
      END;
'P' : BEGIN
          LINE;
          TWOSIDE;
          TWOSIDE;
          LINE;
          SIDE;
          SIDE;
          SIDE
      END;
```

```
          'Q' : BEGIN
                   LINE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   LINE;
                   FARSIDE
                END;
          'R' : BEGIN
                   LINE;
                   TWOSIDE;
                   TWOSIDE;
                   LINE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE
                END;

          'S' : BEGIN
                   LINE;
                   SIDE;
                   SIDE;
                   LINE;
                   FARSIDE;
                   FARSIDE;
                   LINE
                END;
          'T' : BEGIN
                   LINE;
                   MIDLINE;
                   MIDLINE;
                   MIDLINE;
                   MIDLINE;
                   MIDLINE;
                   MIDLINE
                END;
          'U' : BEGIN
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   LINE
                END;
          'V' : BEGIN
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TWOSIDE;
                   TOPV
                END;
```

```
            'W' : DRAWMW(FALSE);

            'X' : BEGIN
                     TOPV;
                     BOTV
                  END;
            'Y' : BEGIN
                     TOPV;
                     MIDLINE;
                     MIDLINE;
                     MIDLINE;
                     MIDLINE
                  END;
            'Z' : BEGIN
                     LINE;
                     BACKDIAG;
                     LINE
                  END
         END;
      END;

   END; (* DRAW *)
BEGIN
   WRITELN;
   WHILE NOT EOLN DO
   BEGIN
      READ(CH);
      WRITELN;
      DRAW;
      WRITELN
   END
END.  (* BLOCK  D.H. 7/22/81 *)
```



# NIM

**The Game.** Nim is a game that has been around for a long time; it is the sort of game one remembers playing during recess in grammar school. Two players take turns picking up a quantity of matches (or toothpicks, or pennies . . . ) from a pile. The object of the game is to leave the last match for your opponent.

In this version of the game, you play against your old pal the computer. In the first game there are 11 matches. You may each pick up 1, 2, or 3 of them at a time. You begin the game by entering your number when you see:

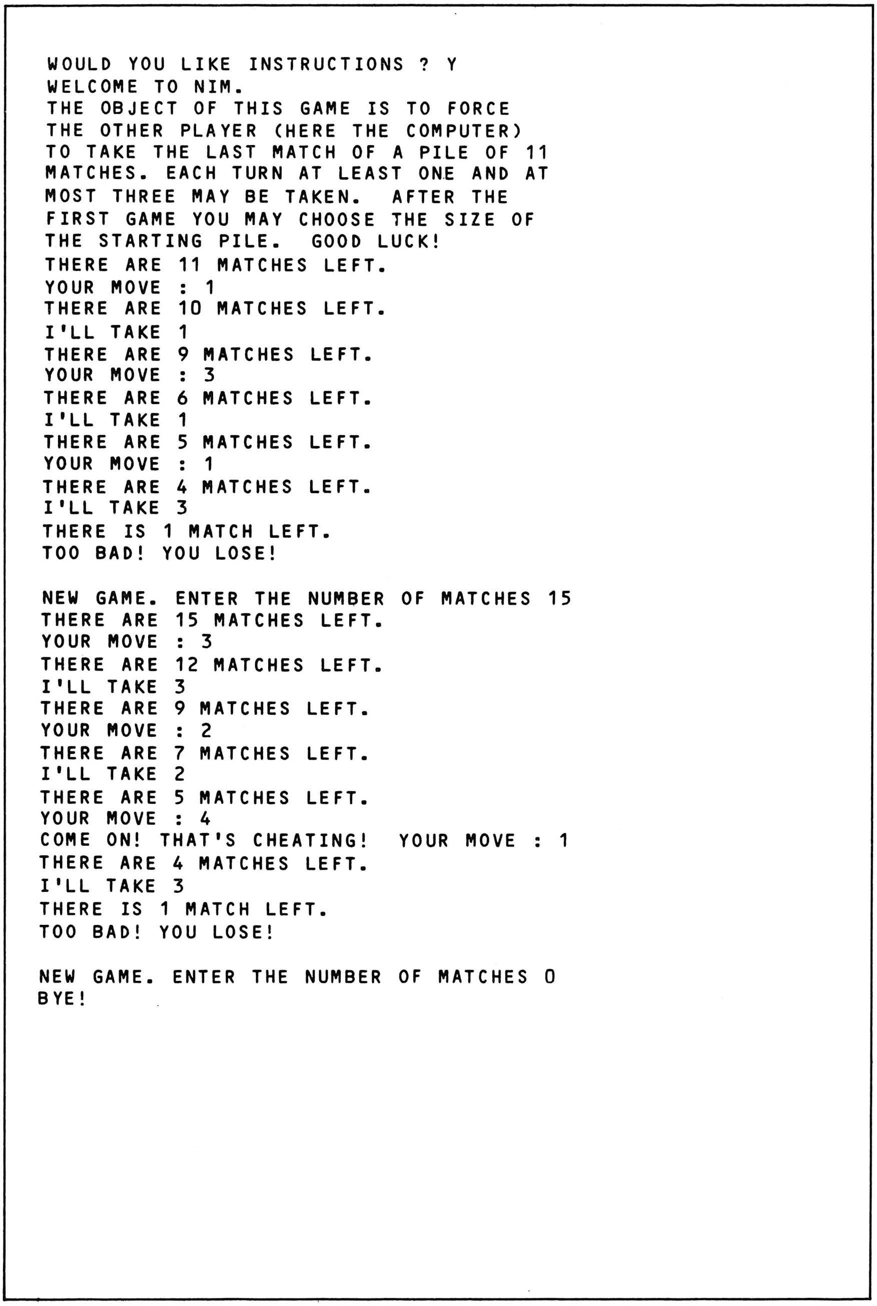
```
YOUR MOVE:
```

The computer's eye is quicker than your hand, so it's no use trying to cheat by picking up more than 3 matches at a time.

The computer will always tell you how many matches it is taking, and at the end of the game it will announce the winner. After the first game you can vary the number of matches in the starting pile if you wish. You will be told:

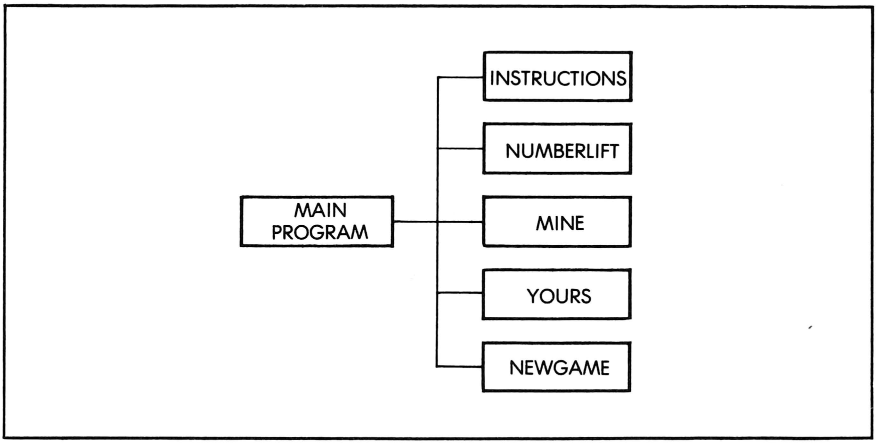
```
NEW GAME. ENTER THE NUMBER OF MATCHES.
```

Keep the number under 25, however; the computer gets bored when the game is too long.



```
WOULD YOU LIKE INSTRUCTIONS ? Y
WELCOME TO NIM.
THE OBJECT OF THIS GAME IS TO FORCE
THE OTHER PLAYER (HERE THE COMPUTER)
TO TAKE THE LAST MATCH OF A PILE OF 11
MATCHES. EACH TURN AT LEAST ONE AND AT
MOST THREE MAY BE TAKEN.  AFTER THE
FIRST GAME YOU MAY CHOOSE THE SIZE OF
THE STARTING PILE.  GOOD LUCK!
THERE ARE 11 MATCHES LEFT.
YOUR MOVE : 1
THERE ARE 10 MATCHES LEFT.
I'LL TAKE 1
THERE ARE 9 MATCHES LEFT.
YOUR MOVE : 3
THERE ARE 6 MATCHES LEFT.
I'LL TAKE 1
THERE ARE 5 MATCHES LEFT.
YOUR MOVE : 1
THERE ARE 4 MATCHES LEFT.
I'LL TAKE 3
THERE IS 1 MATCH LEFT.
TOO BAD! YOU LOSE!

NEW GAME. ENTER THE NUMBER OF MATCHES 15
THERE ARE 15 MATCHES LEFT.
YOUR MOVE : 3
THERE ARE 12 MATCHES LEFT.
I'LL TAKE 3
THERE ARE 9 MATCHES LEFT.
YOUR MOVE : 2
THERE ARE 7 MATCHES LEFT.
I'LL TAKE 2
THERE ARE 5 MATCHES LEFT.
YOUR MOVE : 4
COME ON! THAT'S CHEATING!  YOUR MOVE : 1
THERE ARE 4 MATCHES LEFT.
I'LL TAKE 3
THERE IS 1 MATCH LEFT.
TOO BAD! YOU LOSE!

NEW GAME. ENTER THE NUMBER OF MATCHES 0
BYE!
```

# The Program.



Before the first game, the main program calls procedure INSTRUCTIONS to give the player a chance to ask for a description of the game. Then the variable LEFT, which always records the number of matches left in the pile, is set to INITIAL (i.e., 11) for the first round.

A **REPEAT/UNTIL** loop takes control and continues looping until the player wants to quit (indicated by an input of 1 or less for the new pile of matches). The BOOLEAN variable MYMOVE is set to FALSE, so that the player always goes first in every round. A call to procedure NUMBERLEFT simply displays the number of matches left in the pile.

Then the play itself is controlled by a nested **REPEAT/UNTIL** loop that continues until the game is over. Depending on the value of MYMOVE, LEFT is decreased via a call to function MINE (for the computer's turn) or YOURS (for the player's turn):

```
IF MYMOVE THEN
   LEFT := LEFT – MINE
ELSE
   LEFT := LEFT – YOURS
```

Function MINE continues the computer's algorithm for deciding how many matches to pick up:

```
MOVE := (LEFT − 1) MOD 4;
```

This formula assures the computer of a win if there are from 2 to 4 matches left on the computer's turn. On the other hand, if the computer anticipates that it is losing (if there are exactly 5 matches left) it only takes one match (maybe the player will make a mistake):

```
IF MOVE = 0 THEN
   MOVE := 1;
```

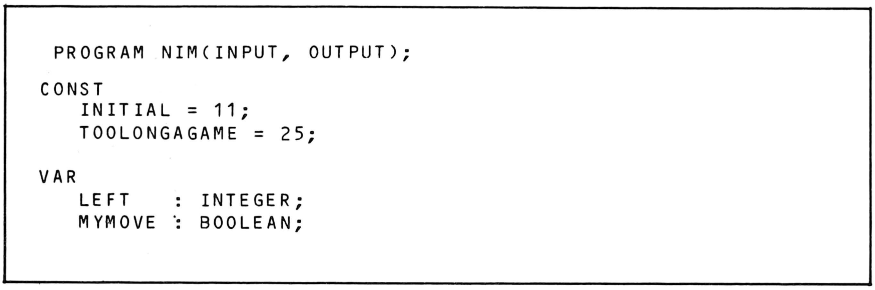
MYMOVE is changed to FALSE (for the player's next turn) and the computer displays its move.

Function YOURS reads the input for the player's move. If the player tries anything funny (like entering a 0 or a 4) an error message is displayed and the new input is read.

The main program then calls NUMBERLEFT again to display the number of matches in the pile. When LEFT = 1, the nested **REPEAT/UNTIL** loop is ended, the winner (determined by the present value of MYMOVE) is announced, and procedure NEWGAME is called.

NEWGAME simply reads the player's input for the new number of matches in the pile. It writes an impatient error message if the player tries to enter a number of 25 or over:

```
WHILE LEFT > TOOLONGAGAME DO
   BEGIN
      WRITELN('I''D GET BORED PLAYING A GAME THAT LONG!');
      WRITE('HOW MANY MATCHES? ');
      READLN(LEFT);
   END;
```

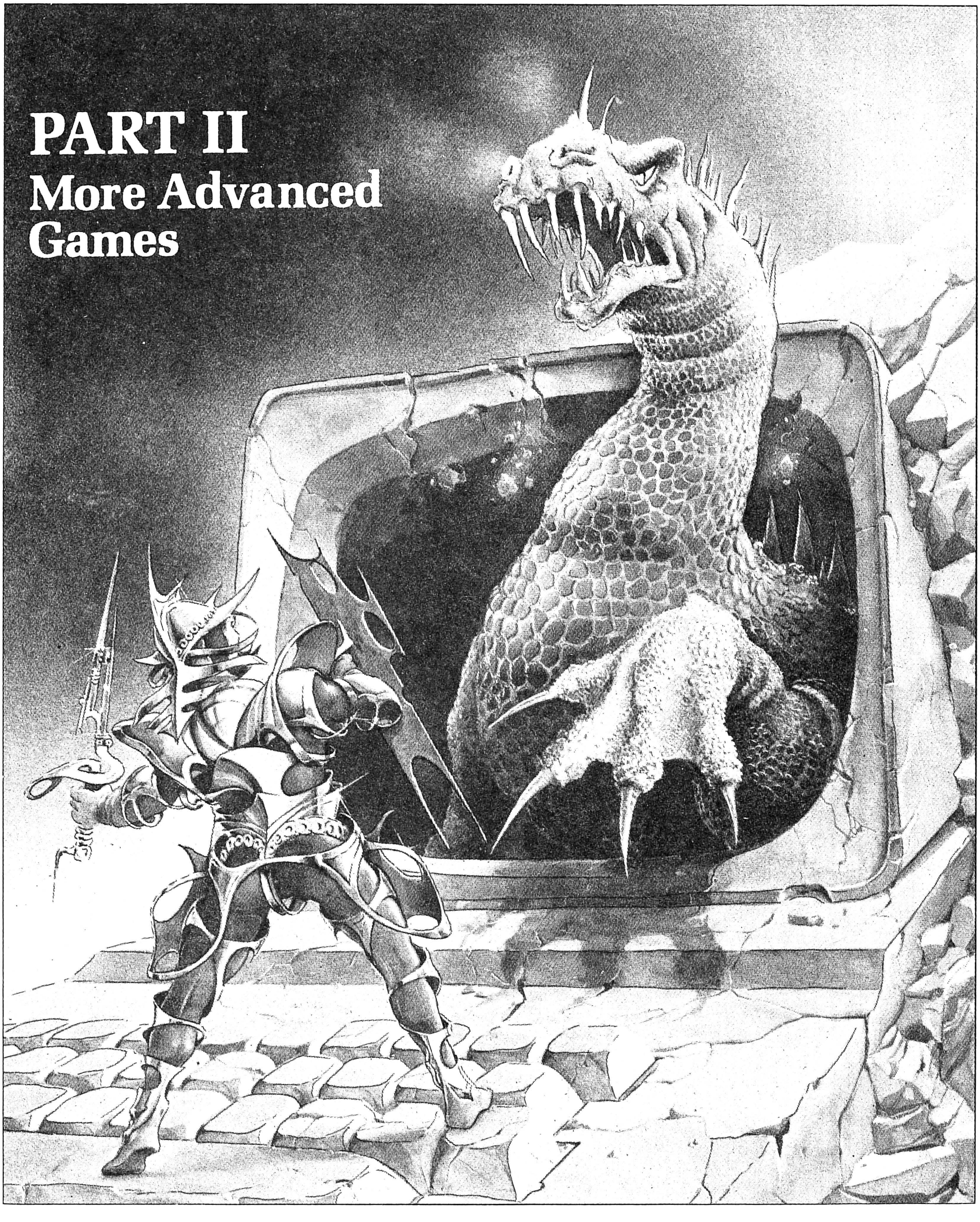


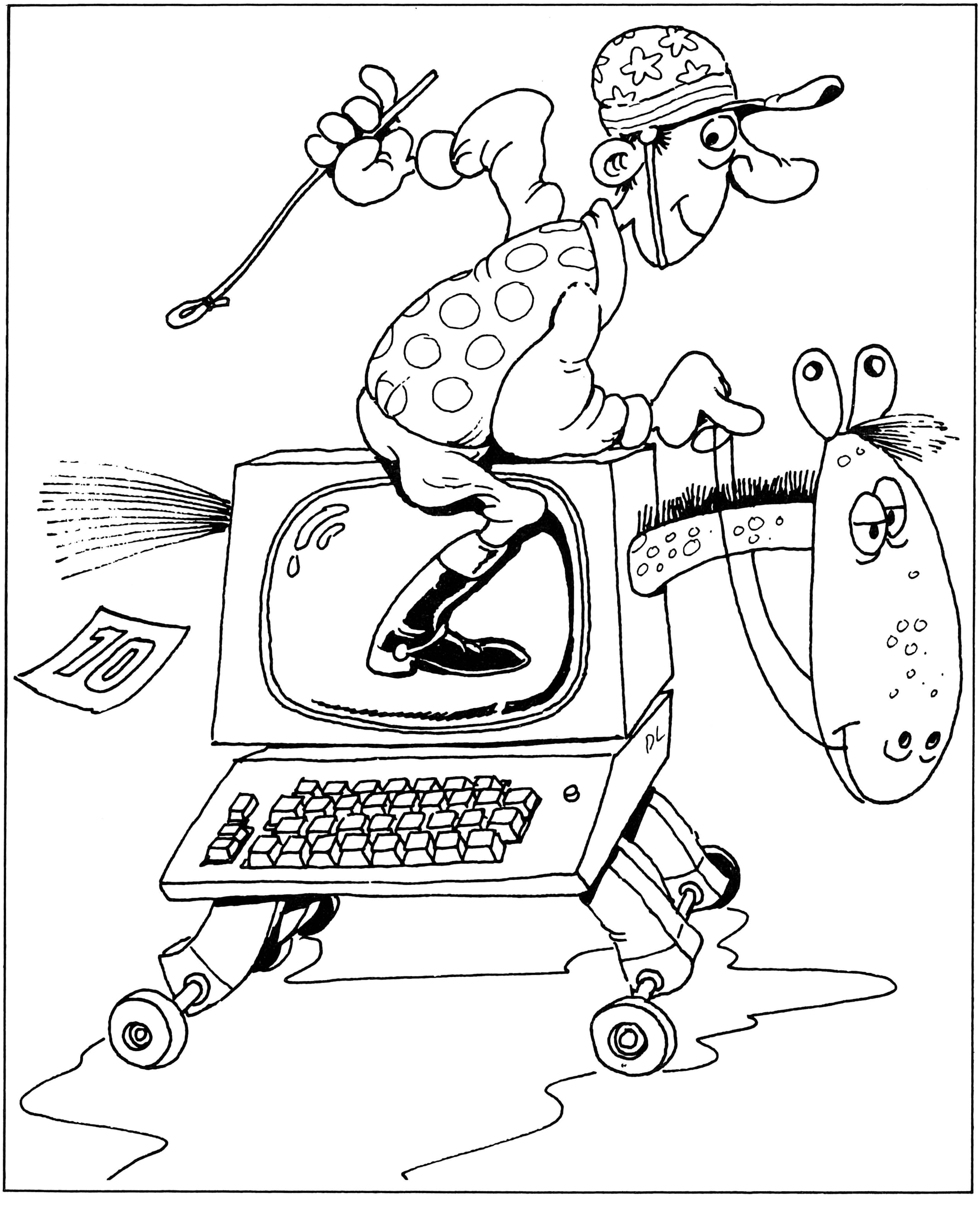
```
 PROGRAM NIM(INPUT, OUTPUT);

CONST
   INITIAL = 11;
   TOOLONGAGAME = 25;

VAR
   LEFT   : INTEGER;
   MYMOVE : BOOLEAN;
```

```
(*
 * ASKS FOR THE NUMBER OF MATCHES TO BE USED
 * IN THE NEXT GAME.  NUMBERS WHICH ARE TOO LARGE
 * FOR A REASONABLY LONG GAME ARE REJECTED.
 *)

PROCEDURE NEWGAME;
BEGIN
   WRITELN;             (* BLANK LINE BETWEEN GAMES     *)
   WRITE('NEW GAME. ENTER THE NUMBER OF MATCHES ');
   READLN(LEFT);
   WHILE LEFT > TOOLONGAGAME DO
   BEGIN
      WRITELN('I''D GET BORED PLAYING A GAME THAT LONG!');
      WRITE('HOW MANY MATCHES ? ');
      READLN(LEFT);
   END;
END; (* NEWGAME *)

(*
 * NUMBERLEFT PRINTS THE NUMBER OF MATCHES REMAINING
 *)

PROCEDURE NUMBERLEFT;
BEGIN
   IF LEFT = 1 THEN
      WRITELN('THERE IS 1 MATCH LEFT.')
   ELSE
      WRITELN('THERE ARE ', LEFT:1, ' MATCHES LEFT.');
END; (* NUMBERLEFT *)

(*
 * THIS PROCEDURE FIGURES THE COMPUTER'S MOVE.  IF IT CAN BE
 * FORCED INTO A LOSS IT CHOOSES ONLY ONE MATCH SO THAT THE
 * PLAYER WILL HAVE MORE CHANCES IN WHICH TO MAKE
 * A MISTAKE.
 *)

FUNCTION MINE : INTEGER;
VAR
   MOVE : INTEGER;
BEGIN
   MOVE := (LEFT - 1) MOD 4;
   IF MOVE = 0 THEN     (* GAME MIGHT BE LOST           *)
      MOVE:= 1;
   MYMOVE := FALSE;     (* NEXT IS THE PLAYER'S TURN    *)
   WRITELN('I''LL TAKE ', MOVE:1);
   MINE := MOVE;
END; (* MINE *)
```

```
FUNCTION YOURS : INTEGER;
VAR
   YOURMOVE : INTEGER;
BEGIN
   WRITE('YOUR MOVE : ');
   READLN(YOURMOVE);
   WHILE (YOURMOVE > 3) OR (YOURMOVE < 1) OR
         (YOURMOVE >= LEFT) DO
   BEGIN
      WRITE('COME ON! THAT'S CHEATING!   YOUR MOVE : ');
      READLN(YOURMOVE);
   END;
   MYMOVE := TRUE;      (* COMPUTER'S TURN NEXT         *)
   YOURS := YOURMOVE;
END; (* YOURS *)

PROCEDURE INSTRUCTIONS;
VAR
   ANSWER : CHAR;
BEGIN
   WRITE('WOULD YOU LIKE INSTRUCTIONS ? ');
   READLN(ANSWER);
   IF ANSWER = 'Y' THEN
   BEGIN
      WRITELN('WELCOME TO NIM.');
      WRITELN('THE OBJECT OF THIS GAME IS TO FORCE');
      WRITELN('THE OTHER PLAYER (HERE THE COMPUTER)');
      WRITELN('TO TAKE THE LAST MATCH OF A PILE OF ', INITIAL:1);
      WRITELN('MATCHES. EACH TURN AT LEAST ONE AND AT');
      WRITELN('MOST THREE MAY BE TAKEN.  AFTER THE');
      WRITELN('FIRST GAME YOU MAY CHOOSE THE SIZE OF');
      WRITELN('THE STARTING PILE.  GOOD LUCK!');
   END;
END; (* INSTRUCTIONS *)

BEGIN           (* MAIN PROGRAM *)
   INSTRUCTIONS;
   LEFT := INITIAL;
   REPEAT
      MYMOVE := FALSE;  (* PLAYER GOES FIRST    *)
      NUMBERLEFT;
      REPEAT
         IF MYMOVE THEN
            LEFT := LEFT - MINE
         ELSE
            LEFT := LEFT - YOURS;
         NUMBERLEFT;
      UNTIL LEFT = 1;
      IF MYMOVE THEN    (* MY TURN WHEN 1 LEFT, I LOST *)
         WRITELN('RATS! YOU WIN!')
      ELSE
         WRITELN('TOO BAD! YOU LOSE!');
      NEWGAME;
   UNTIL LEFT <= 1;
   WRITELN('BYE!');
END.
```

# PART II
# More Advanced Games




10
DL

# HORSERACE

**The Game.** Up to ten players may compete in this game. Each player has twelve horses in a race. When a horse reaches the finish line, the player wins and the game ends. To move the horses down the track, the computer first simulates a roll of dice. Then the player elects to move the horse represented by the sum of the dice or the two horses represented by the individual dice. For example, if a roll of dice produces a 3 and a 6, the player can move his 9th horse one length, or his 3rd and 6th horses one length each.

Certain numbers occur more frequently than others when dice are rolled. There are, for instance, six ways to make the number 6, but only two ways to make a 3. For this reason, the horses are handicapped. They do not all have the same number of lengths from start to finish. For example, the track for the 6th horse is ten lengths, but the track for the 11th horse is only two.

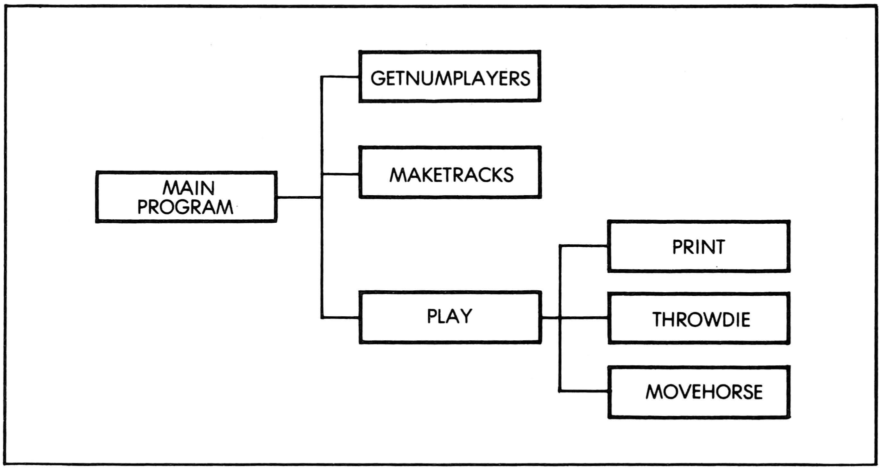
At the start of a turn, a player may ask for a display of all the horses for each player by typing a P. This information helps the player decide which horses to move.

```
HOW MANY PLAYERS? 2
PLAYER NUMBER 1
YOU ROLLED A 1 AND A 6 WITH A TOTAL OF 7
WHICH DO YOU WANT? 1
PLAYER NUMBER 2
YOU ROLLED A 1 AND A 4 WITH A TOTAL OF 5
WHICH DO YOU WANT? 5
PLAYER NUMBER 1
YOU ROLLED A 6 AND A 2 WITH A TOTAL OF 8
WHICH DO YOU WANT? 2
PLAYER NUMBER 2
YOU ROLLED A 2 AND A 1 WITH A TOTAL OF 3
WHICH DO YOU WANT? 3
PLAYER NUMBER 1
YOU ROLLED A 1 AND A 4 WITH A TOTAL OF 5
WHICH DO YOU WANT? 1
PLAYER NUMBER 2
YOU ROLLED A 4 AND A 2 WITH A TOTAL OF 6
WHICH DO YOU WANT? 6
PLAYER NUMBER 1P
PLAYER NUMBER 1

 1 -  ( ) (*) ( ) ( )
 2 -  (*) ( ) ( ) ( ) ( ) ( )
 3 -  ( ) ( ) ( ) ( ) ( ) ( )
 4 -  (*) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 5 -  ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 6 -  ( ) (*) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 7 -  ( ) ( ) ( ) ( ) ( ) ( )
 8 -  ( ) ( ) ( ) ( ) ( ) ( )
 9 -  ( ) ( ) ( ) ( )
10 -  ( ) ( ) ( ) ( )
11 -  ( ) ( )
12 -  ( ) ( )

PLAYER NUMBER 2

 1 -  ( ) ( ) ( ) ( )
 2 -  ( ) ( ) ( ) ( ) ( ) ( )
 3 -  (*) ( ) ( ) ( ) ( ) ( )
 4 -  ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 5 -  (*) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 6 -  (*) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 7 -  ( ) ( ) ( ) ( ) ( ) ( )
 8 -  ( ) ( ) ( ) ( ) ( ) ( )
 9 -  ( ) ( ) ( ) ( )
10 -  ( ) ( ) ( ) ( )
11 -  ( ) ( )
12 -  ( ) ( )
```

```
YOU ROLLED A 2 AND A 3 WITH A TOTAL OF 5
WHICH DO YOU WANT? 2
PLAYER NUMBER 2
YOU ROLLED A 6 AND A 5 WITH A TOTAL OF 11
WHICH DO YOU WANT? 11
PLAYER NUMBER 1
YOU ROLLED A 2 AND A 2 WITH A TOTAL OF 4
WHICH DO YOU WANT? 2
PLAYER NUMBER 2
YOU ROLLED A 4 AND A 1 WITH A TOTAL OF 5
WHICH DO YOU WANT?
5
PLAYER NUMBER 1
YOU ROLLED A 4 AND A 1 WITH A TOTAL OF 5
WHICH DO YOU WANT? 4
PLAYER NUMBER 2
YOU ROLLED A 6 AND A 1 WITH A TOTAL OF 7
WHICH DO YOU WANT? 7
PLAYER NUMBER 1
YOU ROLLED A 5 AND A 5 WITH A TOTAL OF 10
WHICH DO YOU WANT? 5
PLAYER NUMBER 2
YOU ROLLED A 3 AND A 4 WITH A TOTAL OF 7
WHICH DO YOU WANT? 7
PLAYER NUMBER 1P
PLAYER NUMBER 1

 1 -  ( ) ( ) (*) ( )
 2 -  ( ) ( ) ( ) (*) ( ) ( )
 3 -  (*) ( ) ( ) ( ) ( ) ( )
 4 -  ( ) (*) ( ) ( ) ( ) ( ) ( ) ( )
 5 -  ( ) (*) ( ) ( ) ( ) ( ) ( ) ( )
 6 -  ( ) (*) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 7 -  ( ) ( ) ( ) ( ) ( ) ( )
 8 -  ( ) ( ) ( ) ( ) ( ) ( )
 9 -  ( ) ( ) ( ) ( )
10 -  ( ) ( ) ( ) ( )
11 -  ( ) ( )
12 -  ( ) ( )

PLAYER NUMBER 2

 1 -  ( ) ( ) ( ) ( )
 2 -  ( ) ( ) ( ) ( ) ( ) ( )
 3 -  (*) ( ) ( ) ( ) ( ) ( )
 4 -  ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 5 -  ( ) (*) ( ) ( ) ( ) ( ) ( ) ( )
 6 -  (*) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
 7 -  ( ) (*) ( ) ( ) ( ) ( )
 8 -  ( ) ( ) ( ) ( ) ( ) ( )
 9 -  ( ) ( ) ( ) ( )
10 -  ( ) ( ) ( ) ( )
11 -  (*) ( )
12 -  ( ) ( )
```

```
YOU ROLLED A 6 AND A 6 WITH A TOTAL OF 12
WHICH DO YOU WANT? 12
PLAYER NUMBER 2
YOU ROLLED A 6 AND A 6 WITH A TOTAL OF 12
WHICH DO YOU WANT? 12
PLAYER NUMBER 1

YOU ROLLED A 6 AND A 3 WITH A TOTAL OF 9
WHICH DO YOU WANT? 3
PLAYER NUMBER 2
YOU ROLLED A 1 AND A 3 WITH A TOTAL OF 4
WHICH DO YOU WANT? 4
PLAYER NUMBER 1
YOU ROLLED A 3 AND A 1 WITH A TOTAL OF 4
WHICH DO YOU WANT? 1

CONGRATULATIONS PLAYER 1
YOUR HORSE 1 HAS WON THE RACE!
```

# The Program.



The main program begins by initializing the BOOLEAN variable WIN to FALSE. (As we will see, the game continues until one of the players' horses crosses the finish line and WIN becomes TRUE.) RANDOMIZE is then called to assure an unpredictable sequence of random numbers when RANDOM is used in function THROWDIE.

Procedures GETNUMPLAYERS and MAKETRACKS are called next. GETNUMPLAYERS asks for the number of players and checks that the number input is within the acceptable range (between 1 and MAXPLAYER). MAKETRACKS in turn sets up twelve tracks for each player in the array TRACK. TRACK is a two-dimensional (MAXPLAYER by MAXHORSES) array of HORSE; HORSE is a record type with two fields:

```
TYPE
  HORSE = RECORD
    HORSELOC: 0..10;
    TOGO: 0..10
  END;
```

TRACK accordingly holds information on twelve horses for each player. Each horse will have a certain position on the track (HORSELOC) and a certain distance still to go to the finish line (TOGO). MAKETRACKS initializes the field TOGO for each horse, setting the handicaps according to the horse's number (from 1 to 12) and the relative probability of the horse's having a chance to move when the dice are rolled. A nested **FOR** loop then initializes each HORSELOC to zero, thus putting all the horses at the starting line.

With all the horses ready to go, the main program can send the first player (CURPLAYER := 1) into the game. The main program's **WHILE** loop calls procedure PLAY for each player in turn until WIN is TRUE; that is, until one of the players' horses has crossed the finish line.

Procedure PLAY is the algorithmic heart of the program. It

- displays the tracks and horses when so requested
- throws the dice
- asks the player to choose either the individual dice or their sum
- moves the corresponding horse(s) and tests for a win.

When PLAY is called, the global variable CURPLAYER is assigned to the formal parameter PLAYER. The procedure first displays the number of the player. The player may then type the letter P at the keyboard. This will cause the procedure PRINT to be called.

PRINT uses three nested **FOR** loops to display the tracks and horses for each player. The inner **FOR** loop writes out the symbols for a single track and horse. (Left and right parentheses represent each length of the track; an asterisk represents the horse on the track.) The middle **FOR** loop executes the inner loop for each of the 12 horses, and the outer loop executes the middle loop for each player.

PLAY then rolls the dice, calling function THROWDIE to assign a random integer between 1 and 6 to the variables DIE1 and DIE2. (The set POSHORSES is created in order to check for errors when the player selects a number.) Then the player is informed of the results of the dice throw, and asked to select either the individual dice or their sum.

After testing for the validity of the player's choice, procedure PLAY calls procedure MOVEHORSE, which advances a horse one length (by adding 1 to the field HORSELOC and subtracting 1 from the field TOGO). MOVEHORSE is called once if the player selected the sum of the dice, twice if the choice was a single die. (However, if the first call results in a winning horse, the second call is skipped.)

Procedure MOVEHORSE tests for a win each time a horse is moved:

```
WIN := (TRACK[PLAYER,HORSENUM].TOGO = 0)
```

The global BOOLEAN variable WIN is set to TRUE when the first horse crosses the finish line (i.e., the field TOGO = 0 for the winning horse). A congratulatory message is printed and the game is over.

```
PROGRAM HORSERACE (INPUT,OUTPUT);

USES APPLESTUFF;

CONST
  MAXPLAYER = 10;
  MAXHORSES = 12;  (* EACH PLAYER HAS 12 HORSES *)

TYPE
  HORSE = RECORD
    HORSELOC : 0..10;
    TOGO     : 0..10;
  END;
```

```
VAR
  TRACK : ARRAY[1..MAXPLAYER, 1..MAXHORSES] OF HORSE;
  NUMPLAYERS : INTEGER;
  CURPLAYER  : 1..MAXPLAYER;
  WIN        : BOOLEAN;
  I, J       : INTEGER;

PROCEDURE GETNUMPLAYERS;

BEGIN
  WRITE ('HOW MANY PLAYERS? ');
  READLN (NUMPLAYERS);
  WHILE (NUMPLAYERS <= 0) OR (NUMPLAYERS > MAXPLAYER) DO
  BEGIN
    WRITELN ('SORRY, ONLY NUMBERS BETWEEN 1 AND ', MAXPLAYER,'ARE ALLOWED');
    WRITE ('HOW MANY PLAYERS? ');
    READLN (NUMPLAYERS)
  END
END;

(*
 *  MAKETRACKS.
 *  INITIALIZES TRACK.TOGO TO THE CORRECT
 *  HANDICAPS, AND TRACK.HORSELOC TO 0 TO START
 *  THE GAME.
 *)


PROCEDURE MAKETRACKS;
BEGIN
  FOR I := 1 TO NUMPLAYERS DO
  BEGIN
    TRACK[I,1].TOGO := 4;
    TRACK[I,2].TOGO := 6;
    TRACK[I,3].TOGO := 6;
    TRACK[I,4].TOGO := 8;
    TRACK[I,5].TOGO := 8;
    TRACK[I,6].TOGO := 10;
    TRACK[I,7].TOGO := 6;
    TRACK[I,8].TOGO := 6;
    TRACK[I,9].TOGO := 4;
    TRACK[I,10].TOGO := 4;
    TRACK[I,11].TOGO := 2;
    TRACK[I,12].TOGO := 2;
    FOR J := 1 TO MAXHORSES DO
      TRACK[I,J].HORSELOC := 0
  END
END;
```

```
PROCEDURE PLAY (PLAYER : INTEGER);
CONST
  DIEMAX = 6;

VAR
  CH : CHAR;
  HORSENUM : INTEGER;
  POSHORSES : SET OF 1..12;
  DIE1, DIE2 : 1..DIEMAX;

(*
 *  THROWDIE.
 *  USES THE RANDOM FUNCTION TO THROW THE DICE.
 *)

FUNCTION THROWDIE: INTEGER;
BEGIN
  THROWDIE := 1 + (RANDOM MOD DIEMAX)
END;

(*
 *  PRINT.
 *  DISPLAYS THE TRACKS.
 *)


PROCEDURE PRINT;

VAR
  PNUM : 1..MAXPLAYER;
  CURHORSE : 1..MAXHORSES;
  LENGTH : 1..10;
  SUM : 1..10;

BEGIN
  FOR PNUM := 1 TO NUMPLAYER DO
  BEGIN
    WRITELN ('PLAYER NUMBER ', PNUM);
    FOR CURHORSE := 1 TO MAXHORSES DO
    BEGIN
      WRITELN;
      WRITE (CURHORSE:2, ' - ');
      SUM := TRACK[PNUM,CURHORSE].TOGO + TRACK[PNUM,CURHORSE].HORSELOC;
      FOR LENGTH := 1 TO SUM DO
        IF LENGTH = TRACK[PNUM,CURHORSE].HORSELOC THEN
          WRITE(' (*)')
        ELSE
          WRITE(' ( )')
    END;
    WRITELN;
    WRITELN
  END;
  WRITELN
  END; (* PRINT *)
```

```
(*
 *  MOVEHORSE.
 *  INCREMENTS HORSELOC, AND DECREMENTS TOGO
 *  TO MOVE THE HORSE.
 *)

PROCEDURE MOVEHORSE (PLAYER,HORSENUM : INTEGER);
BEGIN
  TRACK[PLAYER,HORSENUM].TOGO := TRACK[PLAYER,HORSENUM].TOGO - 1;
  TRACK[PLAYER,HORSENUM].HORSELOC := TRACK[PLAYER,HORSENUM].HORSELOC + 1;
  WIN := (TRACK[PLAYER,HORSENUM].TOGO = 0);
  IF WIN THEN

  BEGIN
    WRITELN;
    WRITELN ('CONGRATULATIONS PLAYER ', PLAYER);
    WRITELN ('YOUR HORSE ', HORSENUM, ' HAS WON THE RACE!')
  END;
END;


BEGIN (* PLAY *)
  WRITE ('PLAYER NUMBER ', PLAYER);
  READLN (CH);
  IF CH = 'P' THEN  (* PLAYER TYPES A P TO REQUEST
                        A DISPLAY OF THE TRACKS *)
  BEGIN
    PRINT;
    READLN
  END;
  DIE1 := THROWDIE;
  DIE2 := THROWDIE;
  POSHORSES := [DIE1, DIE2, (DIE1 + DIE2)];
  WRITE ('YOU ROLLED A ', DIE1, ' AND A ', DIE2);
  WRITELN (' WITH A TOTAL OF ', (DIE1 + DIE2));
  WRITE ('WHICH DO YOU WANT? ');
  READLN (HORSENUM);
  WHILE NOT (HORSENUM IN POSHORSES) DO
  BEGIN
    WRITELN ('SORRY, BUT ONLY ', DIE1, ', ', DIE2, ', OR ', (DIE1 + DIE2));
    WRITELN (' IS ALLOWED');
    WRITE ('WHICH DO YOU WANT? ');
    READLN (HORSENUM)
  END;
  IF HORSENUM = (DIE1 + DIE2) THEN
    MOVEHORSE (PLAYER, (DIE1 + DIE2))
  ELSE
  BEGIN
    MOVEHORSE (PLAYER, DIE1);
    IF NOT WIN THEN
      MOVEHORSE (PLAYER, DIE2)
    END
  END; (* PLAY *)
```

```
BEGIN (* MAIN PROGRAM *)
  WIN := FALSE;
  GETNUMPLAYERS;
  MAKETRACKS;
  CURPLAYER := 1;
  WHILE NOT WIN DO    (* WIN IS SET TO TRUE IN
                         PROCEDURE MOVEHORSE. *)
  BEGIN
    RANDOMIZE;
    PLAY (CURPLAYER);
    IF CURPLAYER = NUMPLAYERS THEN
      CURPLAYER := 1 (* GO BACK TO THE FIRST
                         PLAYER WHEN EVERYONE HAS HAD A TURN *)
    ELSE
      CURPLAYER := CURPLAYER + 1  (* NEXT PLAYER *)
  END;
  WRITELN;
  WRITELN
END.  (* HORSERACE,  D.H. 7/17/81*)
```

# KISMET

**The Game.** This a game of chance and strategy. The chance is in the roll of the dice; the strategy is in deciding how to score each roll to obtain the greatest number of points. Up to four players may compete. A game consists of six rounds, and the winner is the player with the highest score at the end of the sixth round.

You have one turn per round. The computer keeps track of the players, and lets you know when your turn comes up. Each turn begins with the throw of five dice. The roll of each die—its *value* (1 to 6) and its *color* (red, green, black)—is displayed for you, at which point you have several options:

- Enter a P to see your own present score table, or enter an E to see all the players' score tables. (You might make this request for any number of reasons: to help you plan your strategy; to intimidate the other players with your high score; or merely to stall for time while you are deciding what to do next.)
- Enter an S to take the throw of the dice as it stands and score with it. (You will then be asked to choose a scoring category.)
- Enter an R to indicate that you wish to *replace* a die or some combination of dice with another throw.

You are allowed to take the replace option up to three times per turn. If you have not decided to score before the third replace roll, you must take what you have got at that point. Use the score table to decide on your strategy:

*Basic Section*

1. 1 point for each ace
2. 2 points for each deuce
3. 3 points for each trey
4. 4 points for each four
5. 5 points for each five
6. 6 points for each six

| | *Kismet Section* | *Score* |
|---|---|---|
| 7. | Two pairs of the same color | total dice |
| 8. | Three of a kind, any color | total dice |
| 9. | Straight (1,2,3,4,5, or 2,3,4,5,6) | 30 points |
| 10. | Flush (all dice the same color) | 35 points |
| 11. | Full house (three of a kind and a pair, any color) | total dice + 15 points |
| 12. | Full house, all dice the same color | total dice + 20 points |
| 13. | Four of a kind, any color | total dice + 25 points |
| 14. | Yarborough: one free turn | total dice |
| 15. | Kismet: five of a kind, any color | total dice + 50 points |

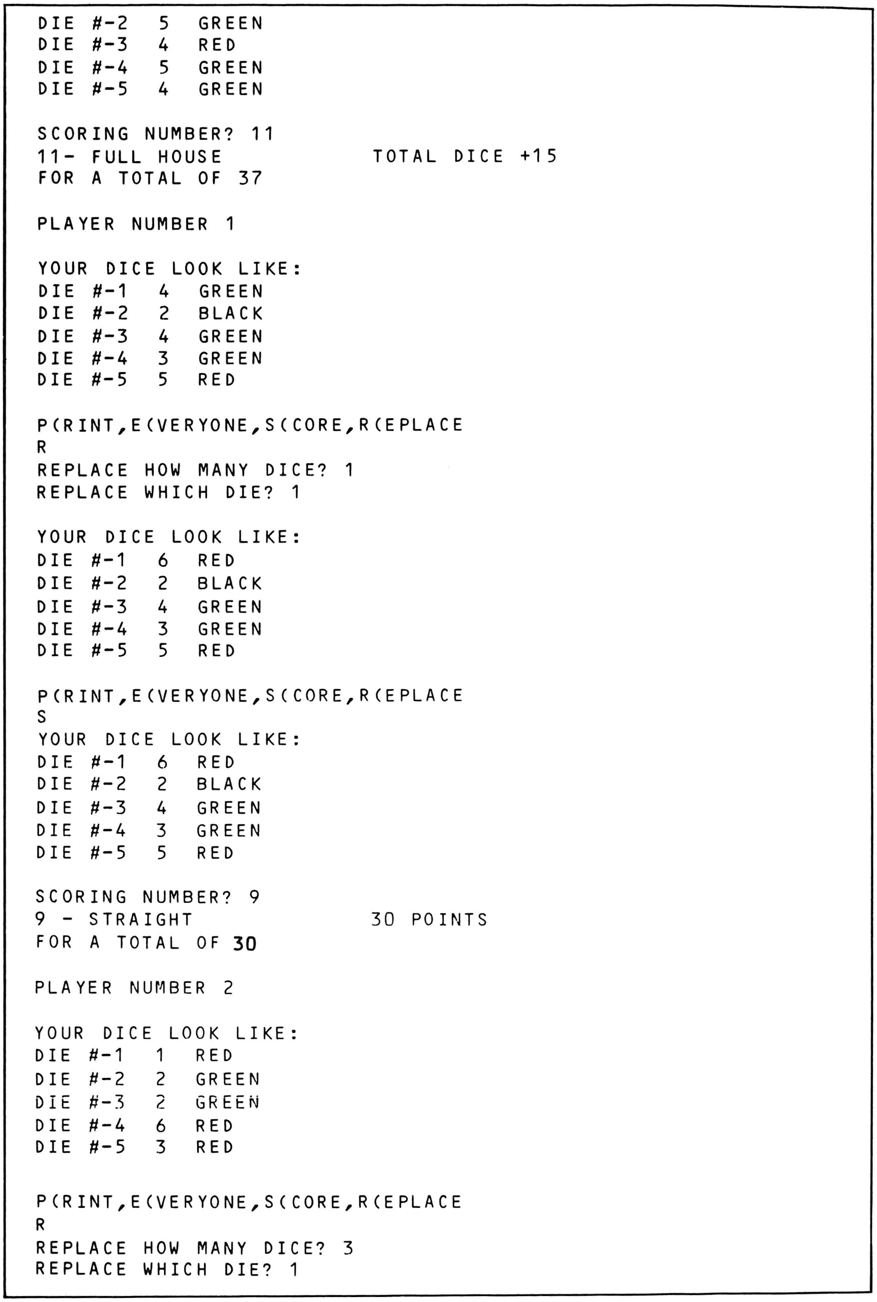
Once you have decided to score, you choose any one of these 15 scoring categories that match your dice. (You may only score one time in each category per game.) Notice that the Kismet Section offers higher points than the Basic Section. However, the Basic Section also gives bonus points for high scores. If you reach one of the following total Basic score ranges, you will automatically be awarded the corresponding number of bonus points:

| *Total Basic Score* | *Bonus* |
|---|---|
| 63 to 70 | 35 points |
| 71 to 77 | 55 points |
| 78 or over | 75 points |

If at some point you find yourself stuck with a bad combination of dice, you may choose the Yarborough category (14 on the score table) once during the game, and receive the total value of your dice as your score. Be careful when you choose the other scoring categories from the Kismet Section. The computer will check up on you to make sure that you really have the scoring combination that you claim to have. Beware: if you try to fool the computer, you get zero points for your efforts!

```
HOW MANY PLAYERS? 2
PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   1   BLACK
DIE #-3   6   BLACK
DIE #-4   5   GREEN
DIE #-5   1   BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 2
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   4   BLACK
DIE #-3   6   BLACK
DIE #-4   5   GREEN
DIE #-5   4   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 3
REPLACE WHICH DIE? 4
```

```
YOUR DICE LOOK LIKE:
DIE #-1  4  GREEN
DIE #-2  4  BLACK
DIE #-3  2  RED
DIE #-4  3  RED
DIE #-5  4  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 3
REPLACE WHICH DIE? 4
YOUR DICE LOOK LIKE:
DIE #-1  4  GREEN
DIE #-2  4  BLACK
DIE #-3  2  GREEN
DIE #-4  2  GREEN
DIE #-5  4  RED


SCORING NUMBER? 11
11- FULL HOUSE          TOTAL DICE +15
FOR A TOTAL OF 31

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1  4  GREEN
DIE #-2  2  RED
DIE #-3  6  RED
DIE #-4  6  RED
DIE #-5  6  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 1
REPLACE WHICH DIE? 2

YOUR DICE LOOK LIKE:
DIE #-1  3  RED
DIE #-2  4  RED
DIE #-3  6  RED
DIE #-4  6  RED
DIE #-5  6  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 1
REPLACE WHICH DIE? 2

YOUR DICE LOOK LIKE:
DIE #-1  6  GREEN
DIE #-2  6  RED
DIE #-3  6  RED
DIE #-4  6  RED
DIE #-5  6  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1  6  GREEN
DIE #-2  6  RED
DIE #-3  6  RED
DIE #-4  6  RED
DIE #-5  6  BLACK

SCORING NUMBER? 15
15- KISMET  5 OF A KIND  TOTAL DICE +50
FOR A TOTAL OF 80

PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1  1  BLACK
DIE #-2  1  RED
DIE #-3  6  RED
DIE #-4  3  BLACK
DIE #-5  5  GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 3
REPLACE WHICH DIE? 3
REPLACE WHICH DIE? 4
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  1  BLACK
DIE #-2  1  RED
DIE #-3  2  RED
DIE #-4  1  RED
DIE #-5  3  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 3
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  1  BLACK
DIE #-2  1  RED
DIE #-3  5  GREEN
DIE #-4  1  RED
DIE #-5  4  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 3
REPLACE WHICH DIE? 5
YOUR DICE LOOK LIKE:
DIE #-1  1  BLACK
DIE #-2  1  RED
DIE #-3  2  RED
DIE #-4  1  RED
DIE #-5  1  GREEN

SCORING NUMBER? 13
13- 4 OF A KIND          TOTAL DICE +25
FOR A TOTAL OF 31

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1  4  RED
DIE #-2  5  GREEN
DIE #-3  4  RED
DIE #-4  5  GREEN
DIE #-5  4  GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
P
1 - ACES    1 FOR EACH ACE                = 0
2 - DEUCES  2 FOR EACH DEUCE              = 0
3 - TREYS   3 FOR EACH TREY               = 0
4 - FOURS   4 FOR EACH FOUR               = 0
5 - FIVES   5 FOR EACH FIVE               = 0
6 - SIXES   6 FOR EACH SIX                = 0

BASIC SECTION TOTALS TO 0

7 - 2 PAIR  SAME COLOR   TOTAL DICE       = 0
8 - 3 OF A KIND          TOTAL DICE       = 0
9 - STRAIGHT             30 POINTS        = 0
10- FLUSH   SAME COLOR   35 POINTS        = 0
11- FULL HOUSE           TOTAL DICE +15   = 0
12- FULL HOUSE SAME COLOR TOTAL DICE +20  = 0
13- 4 OF A KIND          TOTAL DICE +25   = 0
14- YARBOROUGH  FREE TURN TOTAL DICE      = 0
15- KISMET  5 OF A KIND  TOTAL DICE +50   = 80

TOTAL GAME VALUE = 80

YOUR DICE LOOK LIKE:
DIE #-1  4  RED
DIE #-2  5  GREEN
DIE #-3  4  RED
DIE #-4  5  GREEN
DIE #-5  4  GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1  4  RED
```



```
DIE #-2   5   GREEN
DIE #-3   4   RED
DIE #-4   5   GREEN
DIE #-5   4   GREEN

SCORING NUMBER? 11
11- FULL HOUSE              TOTAL DICE +15
FOR A TOTAL OF 37

PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   2   BLACK
DIE #-3   4   GREEN
DIE #-4   3   GREEN
DIE #-5   5   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 1

YOUR DICE LOOK LIKE:
DIE #-1   6   RED
DIE #-2   2   BLACK
DIE #-3   4   GREEN
DIE #-4   3   GREEN
DIE #-5   5   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1   6   RED
DIE #-2   2   BLACK
DIE #-3   4   GREEN
DIE #-4   3   GREEN
DIE #-5   5   RED

SCORING NUMBER? 9
9 - STRAIGHT                30 POINTS
FOR A TOTAL OF 30

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1   1   RED
DIE #-2   2   GREEN
DIE #-3   2   GREEN
DIE #-4   6   RED
DIE #-5   3   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 3
REPLACE WHICH DIE? 1
REPLACE WHICH DIE? 4
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  6  GREEN
DIE #-2  2  GREEN
DIE #-3  2  GREEN
DIE #-4  5  GREEN
DIE #-5  6  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  6  GREEN
DIE #-2  2  GREEN
DIE #-3  2  GREEN
DIE #-4  5  GREEN
DIE #-5  4  GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1  6  GREEN
DIE #-2  2  GREEN
DIE #-3  2  GREEN
DIE #-4  5  GREEN
DIE #-5  4  GREEN

SCORING NUMBER? 10
10- FLUSH   SAME COLOR   35 POINTS
FOR A TOTAL OF 35

PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1  6  BLACK
DIE #-2  5  RED
DIE #-3  5  RED
DIE #-4  3  BLACK
DIE #-5  1  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 3
REPLACE WHICH DIE? 1
REPLACE WHICH DIE? 4
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  4  RED
DIE #-2  5  RED
DIE #-3  5  RED
DIE #-4  6  GREEN
DIE #-5  4  GREEN
```

```
P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 3
REPLACE WHICH DIE? 1
REPLACE WHICH DIE? 4
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1   5   GREEN
DIE #-2   5   RED
DIE #-3   5   RED
DIE #-4   5   RED
DIE #-5   4   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
P
1 - ACES     1 FOR EACH ACE                = 0
2 - DEUCES   2 FOR EACH DEUCE              = 0
3 - TREYS    3 FOR EACH TREY               = 0
4 - FOURS    4 FOR EACH FOUR               = 0
5 - FIVES    5 FOR EACH FIVE               = 0
6 - SIXES    6 FOR EACH SIX                = 0

BASIC SECTION TOTALS TO 0

7 - 2 PAIR   SAME COLOR   TOTAL DICE       = 0
8 - 3 OF A KIND           TOTAL DICE       = 0
9 - STRAIGHT              30 POINTS        = 30
10- FLUSH    SAME COLOR   35 POINTS        = 0
11- FULL HOUSE            TOTAL DICE +15   = 31
12- FULL HOUSE SAME COLOR TOTAL DICE +20   = 0
13- 4 OF A KIND           TOTAL DICE +25   = 31
14- YARBOROUGH   FREE TURN TOTAL DICE      = 0
15- KISMET   5 OF A KIND  TOTAL DICE +50   = 0

TOTAL GAME VALUE = 92

YOUR DICE LOOK LIKE:
DIE #-1   5   GREEN
DIE #-2   5   RED
DIE #-3   5   RED
DIE #-4   5   RED
DIE #-5   4   RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 5
YOUR DICE LOOK LIKE:
DIE #-1   5   GREEN
DIE #-2   5   RED
DIE #-3   5   RED
DIE #-4   5   RED
DIE #-5   6   GREEN
```

```
SCORING NUMBER? 5
5 - FIVES    5 FOR EACH FIVE
FOR A TOTAL OF 20

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   1   GREEN
DIE #-3   3   RED
DIE #-4   5   GREEN
DIE #-5   5   GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 3

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   1   GREEN
DIE #-3   3   BLACK
DIE #-4   5   GREEN
DIE #-5   5   GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 3

YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   1   GREEN
DIE #-3   2   GREEN
DIE #-4   5   GREEN
DIE #-5   5   GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1   4   GREEN
DIE #-2   1   GREEN
DIE #-3   2   GREEN
DIE #-4   5   GREEN
DIE #-5   5   GREEN

SCORING NUMBER? 10
SCORING NUMBER? 14
14- YARBOROUGH   FREE TURN TOTAL DICE
FOR A TOTAL OF 17

PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1   3   BLACK
DIE #-2   4   RED
DIE #-3  2  RED
DIE #-4  4  RED
DIE #-5  4  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1  3  BLACK
DIE #-2  4  RED
DIE #-3  2  RED
DIE #-4  4  RED
DIE #-5  4  RED

SCORING NUMBER? 8
8 - 3 OF A KIND          TOTAL DICE
FOR A TOTAL OF 17

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1  5  GREEN
DIE #-2  1  GREEN
DIE #-3  1  GREEN
DIE #-4  2  RED
DIE #-5  1  BLACK

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1  5  GREEN
DIE #-2  1  GREEN
DIE #-3  1  GREEN
DIE #-4  2  RED
DIE #-5  1  BLACK

SCORING NUMBER? 8
8 - 3 OF A KIND          TOTAL DICE
FOR A TOTAL OF 10

PLAYER NUMBER 1

YOUR DICE LOOK LIKE:
DIE #-1  3  RED
DIE #-2  2  RED
DIE #-3  6  RED
DIE #-4  4  BLACK
DIE #-5  1  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
P
1 - ACES     1 FOR EACH ACE               = 0
2 - DEUCES   2 FOR EACH DEUCE             = 0
3 - TREYS    3 FOR EACH TREY              = 0
4 - FOURS    4 FOR EACH FOUR              = 0
5 - FIVES    5 FOR EACH FIVE              = 20
6 - SIXES    6 FOR EACH SIX               = 0
```

```
BASIC SECTION TOTALS TO 20

7 - 2 PAIR   SAME COLOR   TOTAL DICE        = 0
8 - 3 OF A KIND           TOTAL DICE        = 17
9 - STRAIGHT              30 POINTS         = 30
10- FLUSH    SAME COLOR   35 POINTS         = 0
11- FULL HOUSE            TOTAL DICE +15    = 31
12- FULL HOUSE SAME COLOR TOTAL DICE +20    = 0
13- 4 OF A KIND           TOTAL DICE +25    = 31
14- YARBOROUGH   FREE TURN TOTAL DICE       = 0
15- KISMET   5 OF A KIND  TOTAL DICE +50    = 0

TOTAL GAME VALUE = 129

YOUR DICE LOOK LIKE:
DIE #-1   3  RED
DIE #-2   2  RED
DIE #-3   6  RED
DIE #-4   4  BLACK
DIE #-5   1  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
S
YOUR DICE LOOK LIKE:
DIE #-1   3  RED
DIE #-2   2  RED
DIE #-3   6  RED
DIE #-4   4  BLACK
DIE #-5   1  RED

SCORING NUMBER? 14
14- YARBOROUGH   FREE TURN TOTAL DICE
FOR A TOTAL OF 16

PLAYER NUMBER 2

YOUR DICE LOOK LIKE:
DIE #-1   3  RED
DIE #-2   4  GREEN
DIE #-3   6  GREEN
DIE #-4   3  GREEN
DIE #-5   4  RED

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 3

YOUR DICE LOOK LIKE:
DIE #-1   3  RED
DIE #-2   4  GREEN
DIE #-3   5  BLACK
DIE #-4   3  GREEN
DIE #-5   4  RED
```

```
P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 2
REPLACE WHICH DIE? 4
REPLACE WHICH DIE? 5

YOUR DICE LOOK LIKE:
DIE #-1  3  RED
DIE #-2  4  GREEN
DIE #-3  5  BLACK
DIE #-4  2  BLACK
DIE #-5  2  GREEN

P(RINT,E(VERYONE,S(CORE,R(EPLACE
R
REPLACE HOW MANY DICE? 1
REPLACE WHICH DIE? 5
YOUR DICE LOOK LIKE:
DIE #-1  3  RED
DIE #-2  4  GREEN
DIE #-3  5  BLACK
DIE #-4  2  BLACK
DIE #-5  3  RED

SCORING NUMBER? 3
3 - TREYS   3 FOR EACH TREY
FOR A TOTAL OF 6

1 - ACES    1 FOR EACH ACE                  = 0
2 - DEUCES  2 FOR EACH DEUCE                = 0
3 - TREYS   3 FOR EACH TREY                 = 0
4 - FOURS   4 FOR EACH FOUR                 = 0
5 - FIVES   5 FOR EACH FIVE                 = 20
6 - SIXES   6 FOR EACH SIX                  = 0

BASIC SECTION TOTALS TO 20

7 - 2 PAIR  SAME COLOR   TOTAL DICE         = 0
8 - 3 OF A KIND          TOTAL DICE         = 17
9 - STRAIGHT             30 POINTS          = 30
10- FLUSH   SAME COLOR   35 POINTS          = 0
11- FULL HOUSE           TOTAL DICE +15     = 31
12- FULL HOUSE SAME COLOR TOTAL DICE +20    = 0
13- 4 OF A KIND          TOTAL DICE +25     = 31
14- YARBOROUGH  FREE TURN TOTAL DICE        = 16
15- KISMET  5 OF A KIND  TOTAL DICE +50     = 0

TOTAL GAME VALUE = 145
1 - ACES    1 FOR EACH ACE                  = 0
2 - DEUCES  2 FOR EACH DEUCE                = 0
3 - TREYS   3 FOR EACH TREY                 = 6
4 - FOURS   4 FOR EACH FOUR                 = 0
5 - FIVES   5 FOR EACH FIVE                 = 0
6 - SIXES   6 FOR EACH SIX                  = 0
```

```
BASIC SECTION TOTALS TO 6

7 - 2 PAIR  SAME COLOR    TOTAL DICE        = 0
8 - 3 OF A KIND           TOTAL DICE        = 10
9 - STRAIGHT              30 POINTS         = 0
10- FLUSH  SAME COLOR     35 POINTS         = 35
11- FULL HOUSE            TOTAL DICE +15    = 37
12- FULL HOUSE SAME COLOR TOTAL DICE +20    = 0
13- 4 OF A KIND           TOTAL DICE +25    = 0
14- YARBOROUGH  FREE TURN TOTAL DICE        = 17
15- KISMET  5 OF A KIND   TOTAL DICE +50    = 80

TOTAL GAME VALUE = 185
```

# The Program.

The main program begins by asking for the number of players and storing the response in NUMPLAYERS. Then procedure INIT is called to perform three initialization tasks: (1) set all the scores for all the players to zero in SCORE; (2) store the score table information in TABLES (an array of STRING); and (3) assign string values to the array COLORS. (We will see how COLORS is used when we look at procedure PRINTDICE.) Note that SCORE is an array of arrays: the inner loop (index J) of INIT goes through the 15 score categories, and the outer loop (index K) increments the player number from 1 to NUMPLAYERS.

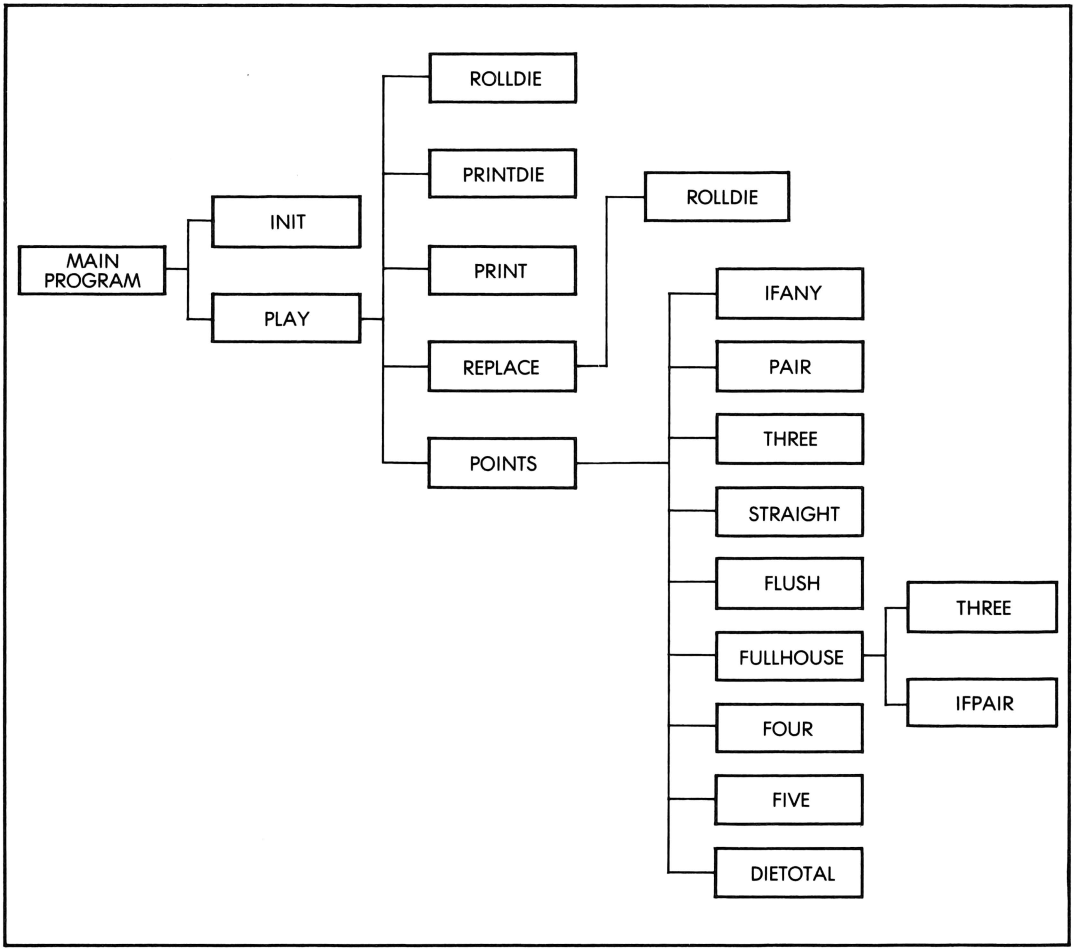
Next the main program begins its series of calls to procedure PLAY. PLAY is called inside a pair of nested **FOR** loops; the inner loop gives each player a turn, and the outer loop controls the number of rounds. Thus, PLAY is called NUMPLAYERS × MAXPLAYS times. Each time PLAY is called, the number of the current player is passed to the formal parameter PLAYER.

PLAY identifies the player:

```
WRITELN('PLAYER NUMBER ',PLAYER);
```

and rolls the dice. Procedure ROLLDIE is then called NUMDICE times to assign values to the array DICE. DICE is an array of records; each record has two fields: VALUE and COLOR.

ROLLDIE calls the RANDOM function twice for each die. The first RANDOM call results in an integer from 1 to 6 for the VALUE field.


MAIN PROGRAM
INIT
PLAY
ROLLDIE
PRINTDIE
PRINT
REPLACE
POINTS
ROLLDIE
IFANY
PAIR
THREE
STRAIGHT
FLUSH
FULLHOUSE
FOUR
FIVE
DIETOTAL
THREE
IFPAIR

The value returned by the second RANDOM call is transformed into an integer from 1 to 3 by the expression:

```
1 + (RANDOM MOD 3)
```

and then used to select a **CASE** label for assigning a value of RED, GREEN, or BLACK to the COLOR field.

Once the dice are rolled, PLAY initializes the variable ROLL to 1. ROLL is used to count the number of times the player has chosen the *replace* option. As long as ROLL is less than 4 (**WHILE** ROLL < 4 **DO** . . . ), PLAY continues to display the dice (by calling procedure PRINTDICE), and to remind the player of the options available:

```
P(RINT),E(VERYONE),S(CORE),R(EPLACE)
```

PRINTDICE is a small, straightforward procedure that displays the value and color of each die. It has one interesting feature—the use of the array COLORS in the WRITELN statement. Recall that COLORS is an array of STRING, indexed by the values of the scalar type DIECOLOR. Thus, COLORS[DICE[I].COLOR] contains one of the string values that were assigned in procedure INIT.

Procedure PRINT is called once if a P is entered; if an E is entered, PRINT is called NUMPLAYERS times. Each time this procedure is called it displays the entire score table, along with the current scores for the player in question. It also tallies the player's total score, adding in any BONUS points that the player has won.

Procedure REPLACE is called if the player enters an R. REPLACE asks the player how many dice to throw again, and then asks for the number of each die that is to be rethrown. To prevent the player from rolling the same die twice, the set USED keeps track of the dice that have already been thrown. The input die number is validated by a pair of nested **REPEAT/UNTIL** loops.

Finally, if the player types an S or runs out of replace options (ROLL = 4), procedure PLAY displays the final roll of the dice (PRINTDICE) and asks the player for a scoring option. The input scoring number is validated (the number must be between 1 and 15, and the scoring option must be one that the player has not chosen before; that is, the score for that option—SCORE[PLAYER][NUM]—must still be zero). When the correct conditions are met, procedure POINTS is called. NUM, the number of the scoring option, is passed to the formal parameter HOW in POINTS.

POINTS first calls procedure DIETOTAL to sum up the dice values. It then uses the value of HOW in a **CASE** statement to select the correct algorithm for the scoring option the player has chosen.

For the Basic Section options (1, 2, 3, 4, 5, 6) the simple procedure IFANY is called to sum up the resulting score.

For the Kismet Section options (7 to 15), one or two of a series of functions are called (PAIR, THREE, STRAIGHT, FLUSH, FULL-

HOUSE, FOUR, and FIVE) to validate the player's claim to a scoring option. All of these functions return BOOLEAN values: if TRUE, then the player has scored correctly, and SUM is calculated accordingly; if FALSE, then the player has erred (fibbed? cheated?) and SUM is set back to zero.

Each of these validation functions is worth investigating to discover the algorithm it uses to test the values of the dice. We will look at one example here—function STRAIGHT. The **VAR** statement of STRAIGHT defines the set HAS. The function begins by initializing HAS as an empty set:

```
HAS := [ ]
```

Then, in a **FOR** loop, each die value is put into the set. Finally, the resulting set is tested: if HAS = [1,2,3,4,5] or if HAS = [2,3,4,5,6], then STRAIGHT returns the value TRUE; otherwise STRAIGHT returns a FALSE.

Notice one last feature of the **CASE** statement in procedure POINTS. Label 14 corresponds to the Yarborough scoring option. No action is taken at all, and the variable SUM thus retains the value of DIETOTAL:

```
14: ;
```

```
(*
 * KISMET GAME
 *)
PROGRAM KISMET (INPUT,OUTPUT);

USES     APPLESTUFF;                (* NEEDED FOR RANDOM FUNCTIONS *)

CONST
   MAXPLAYERS    =  4;              (* MAXIMUM NUMBER OF PLAYERS *)
   SCORES        =  15;             (* 15 WAYS TO MAKE POINTS *)
   NUMDICE       =  5;              (* THE NUMBER OF DICE IN THE GAME *)
   MAXPLAYS      =  6;              (* NUMBER OF PLAYS IN A GAME *)
   DIEMAX        =  6;              (* DIE VALUES GO FROM 1-6 *)

TYPE
   DIECOLOR      = (RED,GREEN,BLACK,WHITE);  (* POSSIBLE COLORS *)
   DIEVALUE      = 1..DIEMAX;       (* POSSIBLE VALUES A DIE CAN HAVE *)
   DIEINDEX      = 1..NUMDICE;      (* INDEX INTO DICE ARRAY  *)

   DIE   =   RECORD                 (* STRUCTURE FOR EACH DIE *)
        COLOR : DIECOLOR;           (* ITS COLOR *)
        VALUE : DIEVALUE            (* ITS VALUE *)
   END;
```

```
    TABLETYPE     = 1..SCORES;     (* POSSIBLE WAYS TO SCORE *)
    SCORETYPE     = ARRAY[TABLETYPE] OF INTEGER;  (* USED FOR SCORING *)

 VAR
    DICE  : ARRAY [DIEINDEX] OF DIE;      (* THE DICE *)
    SCORE : ARRAY [1..MAXPLAYERS] OF SCORETYPE;   (* KEEP TRACK OF SCORES *)
    NUMPLAYERS   : INTEGER;       (* NUMBER OF PLAYERS IN GAME *)
    CURGAME      : 1..MAXPLAYS;   (* CURRENT PLAY NUMBER *)
    CH   : CHAR;                  (* INPUT CHARACTER *)
    COLORS : ARRAY [DIECOLOR] OF STRING; (* WILL CONTAIN 'RED',... *)
    TABLES : ARRAY[TABLETYPE] OF STRING; (* HOW SCORING IS DONE *)
    J : 1..MAXPLAYERS;

 (*
  * PRINT
  * PRINTS OUT A GIVEN PLAYERS POINTS
  *)
 PROCEDURE PRINT(PLAYER:INTEGER);
 VAR
    SUM : INTEGER;              (* POINT SUM *)
    BONUS : INTEGER;            (* BONUS POINTS *)
    I : TABLETYPE;
 BEGIN
    SUM := 0;
    FOR I := 1 TO 6 DO
    BEGIN
       SUM := SUM + SCORE[PLAYER][I];
       WRITELN(TABLES[I],' = ',SCORE[PLAYER][I])
    END;
    BONUS := 0;
    IF SUM >= 63 THEN
       IF SUM <= 70 THEN
          BONUS := 35
       ELSE
          IF SUM <= 77 THEN
             BONUS := 55
          ELSE
             BONUS := 75;
    WRITELN;
    WRITELN('BASIC SECTION TOTALS TO ',SUM);
    IF BONUS <> 0 THEN
    BEGIN
       WRITELN('WITH ADDED BONUS OF ',BONUS);
       WRITELN('GRAND TOTAL FOR BASIC SECTION ',SUM+BONUS)
    END;
    WRITELN;
    SUM := SUM + BONUS;
    FOR I := 7 TO SCORES DO
    BEGIN
       SUM := SUM + SCORE[PLAYER][I];
       WRITELN(TABLES[I],' = ',SCORE[PLAYER][I])
    END;
    WRITELN;
    WRITELN('TOTAL GAME VALUE = ',SUM+BONUS)

 END;
```

```
(*
 * PLAY
 * GIVEN A PLAYER, IT WILL ROLL THE DICE,
 * PRINT OUT THE BOARD, KEEP SCORE, AND DO
 * ANYTHING ELSE THAT NEEDS TO BE DONE
 *)
PROCEDURE   PLAY(PLAYER:INTEGER);
VAR
   NUM  : INTEGER;                 (* INPUT NUMBER FROM USER *)
   ROLL : 1..4;                    (* ROLL WE ARE ON *)
   CH   : CHAR;                    (* INPUT CHARACTER *)
   I    : 1..MAXPLAYERS;           (* INDEX FOR PRINTING EVERYONE'S BOARD *)
   DIENUM : DIEINDEX;              (* USED IN ROLLING THE DICE *)

   (*
    * ROLLDIE
    * WILL RETURN A RANDOM NUMBER, AND A RANDOM
    * COLOR FOR A DIE
    *)
   PROCEDURE ROLLDIE(VAR DICE : DIE);
   BEGIN
      DICE.VALUE := 1+(RANDOM MOD DIEMAX);
      CASE (1+ (RANDOM MOD 3)) OF
         1: DICE.COLOR := RED;
         2: DICE.COLOR := GREEN;
         3: DICE.COLOR := BLACK
      END
   END;

   (*
    * DIETOTAL
    * RETURNS THE TOTAL OF THE SUM OF ALL THE
    * DICE.
    *)
   FUNCTION DIETOTAL:INTEGER;
   VAR
      I : DIEINDEX;
      SUM : INTEGER;               (* TEMPORARY SUM *)
   BEGIN
      SUM := 0;
      FOR I := 1 TO NUMDICE DO
         SUM := SUM + DICE[I].VALUE;
      DIETOTAL := SUM
   END;
   (*
    * POINTS
    * GIVES OUT POINTS TO A PLAYER
    * THE WAY HE (SHE) ASKED. FOR EXAMPLE, IF THE VARIABLE
    * HOW = 8 THEN THE PLAYER HAS ASKED FOR 3-OF-A-KIND
    *)
   PROCEDURE POINTS(HOW:TABLETYPE);
   VAR
      I,J,K  : INTEGER;            (* INDICES INTO DICE ARRAY *)
      RETURN : BOOLEAN;            (* USED BY SUBPROCEDURES *)
      SUM : INTEGER;               (* POINT SUM *)
```

```
(*
 * IFANY
 * SCORES POINTS IF ANY CORRECT
 * DIE VALUES ARE SHOWN
 *)
PROCEDURE IFANY;
BEGIN
   SUM := 0;
   FOR I := 1 TO NUMDICE DO
      IF DICE[I].VALUE = HOW THEN
         SUM := SUM + HOW
END;

(*
 * PAIR
 * SEES IF THERE ARE 2 DIFFRENT PAIRS,
 * WITH THE 2 COMPONENT OF EACH PAIR HAVING
 * THE SAME COLOR
 *)
FUNCTION PAIR:BOOLEAN;
VAR
   NUMPAIRS: 0..3;                (* NUMBER OF PAIRS FOUND *)
BEGIN
   NUMPAIRS := 0;
   FOR I := 1 TO NUMDICE -1 DO
      FOR J := I+1 TO NUMDICE DO
         IF (DICE[I].VALUE = DICE[J].VALUE) AND
            (DICE[I].COLOR = DICE[J].COLOR) THEN
         BEGIN
            NUMPAIRS := NUMPAIRS +1;
            DICE[I].COLOR := WHITE;
            DICE[J].COLOR := WHITE        (* MAKE SURE NEITHER
                                           * ONE IS REUSED IN A TEST
                                           *)
         END;
   PAIR := NUMPAIRS >= 2
END;

(*
 * THREE
 * RETURNS WHETHER OR NOT THERE IS A THREE OF A KIND IN THE DICE
 *)
FUNCTION THREE:BOOLEAN;
BEGIN
   THREE := FALSE;

   (*
    * SIMPLY ROLL THROUGH ALL COMBINATIONS OF THREE
    * AND SEE IF ANY ARE ALL EQUAL
    *)
   FOR I := 1 TO (NUMDICE - 2) DO
      FOR J := I+1 TO (NUMDICE -1) DO
         FOR K := I+2 TO NUMDICE DO
            IF (DICE[I].VALUE = DICE[J].VALUE) AND
             (DICE[J].VALUE = DICE[K].VALUE) THEN
               THREE := TRUE
END;
```

```
(*
 * STRAIGHT
 * RETURNS WHETHER OR NOT THERE IS A STRAIGHT IN THE DICE
 *)
FUNCTION STRAIGHT:BOOLEAN;
VAR
   HAS : SET OF 1..6;      (* THE DICE PUT IN A SET *)
BEGIN
   (* PUT THE DICE INTO THE SET *)
   HAS := [];
   FOR I := 1 TO NUMDICE DO
      HAS := HAS + [DICE[I].VALUE];
   STRAIGHT := (HAS = [1,2,3,4,5]) OR (HAS = [2,3,4,5,6])
END;

(*
 * FLUSH
 * WHETHER OR NOT THERE IS A FLUSH
 *)

FUNCTION FLUSH:BOOLEAN;
BEGIN
   FLUSH := TRUE;
   FOR I := 1 TO (NUMDICE -1) DO
      IF DICE[I].COLOR <> DICE[I+1].COLOR THEN
         FLUSH := FALSE
END;

(*
 * FULLHOUSE
 * IF THERE IS A FULLHOUSE IN THE DICE
 *)

FUNCTION FULLHOUSE:BOOLEAN;

   (*
    * IFPAIR
    * IF THERE IS A PAIR (BUT NOT 3 OR > OF
    * A KIND)
    *)
   FUNCTION IFPAIR:BOOLEAN;
   VAR
      TMP : ARRAY[1..DIEMAX] OF 0..NUMDICE; (* NUMBER OF EACH POSSIBILITY *)
   BEGIN
      (*
       * ZERO OUT THE ARRAY
       *)
      FOR I := 1 TO DIEMAX DO
         TMP[I] := 0;

      (*
       * COUNT UP THE NUMBER OF
       * EACH VALUE
       *)
      FOR I := 1 TO NUMDICE DO
         TMP[DICE[I].VALUE] := TMP[DICE[I].VALUE] + 1;
      (*
       * SEE IF ANY IS EXACTLY 2
       *)
      FOR I := 1 TO DIEMAX DO
         RETURN := RETURN OR (TMP[I] = 2);

      IFPAIR := RETURN
   END;
   (*
    * FULLHOUSE
    *)
   BEGIN
      FULLHOUSE := THREE AND IFPAIR
   END;

   (*
    * 4 OF A KIND
    *)
   FUNCTION FOUR:BOOLEAN;
   VAR
      COUNTER : INTEGER;
      J : 1..DIEMAX;


   BEGIN
     RETURN := FALSE;
     FOR J := 1 TO DIEMAX DO
       BEGIN
         COUNTER := 0;
         FOR I := 1 TO NUMDICE DO
           IF (DICE[I].VALUE = J) THEN
             COUNTER := COUNTER + 1;
         IF (COUNTER = 4) THEN
           RETURN := TRUE
       END;
     FOUR := RETURN
   END;  (* FOUR, D.H. 7/19/81 *)

   (*
    * FIVE
    * IF THERE IS A FIVE OF A KIND
    *)
   FUNCTION FIVE:BOOLEAN;

    BEGIN
       FIVE := TRUE;
       FOR I:= 1 TO (NUMDICE -1) DO
          IF DICE[I].VALUE <> DICE[I+1].VALUE THEN
             FIVE := FALSE
    END;

(* POINTS *)
BEGIN
   SUM := DIETOTAL;      (*  POINTS TO SUM OF DICE *)
   RETURN := FALSE;
   CASE HOW OF
      1,2,3,4,5,6: IFANY;   (* IF ANY 1'S, 2'S, 3'S ...   *)

      7: IF NOT PAIR THEN
            SUM := 0;

      8: IF NOT THREE THEN
            SUM := 0;       (* NO THREE-OF-A-KINDS *)

      9: IF STRAIGHT THEN
            SUM := 30       (* 30 POINTS FOR A STRAIGHT *)
         ELSE
            SUM := 0;       (* NO STRAIGHT *)

      10: IF FLUSH THEN
            SUM := 35       (* 35 POINTS FOR A FLUSH *)
          ELSE
            SUM := 0;       (* NO FLUSH *)

      11: IF FULLHOUSE THEN (* 15 EXTRA POINTS FOR A FULLHOUSE *)
            SUM := SUM +15
          ELSE
            SUM := 0;       (* NO FULLHOUSE *)

      12: IF FULLHOUSE AND FLUSH THEN
            SUM := SUM + 20 (* 20 EXTRA POINTS FOR A FULLHOUSE
                                  OF THE SAME COLOR            *)
            ELSE
            SUM := 0;       (* NO FULLHOUSE OF THE SAME COLOR *)

      13:
          IF FOUR THEN
            SUM := SUM + 25 (* 25 EXTRA POINTS FOR FOUR-OF-A-KIND *)
          ELSE
            SUM := 0;       (* NO FOUR-OF-A-KIND *)

      14: ;                 (* FREE MOVE, GET SUM OF DIE *)

      15: IF FIVE THEN
            SUM := SUM + 50 (* 50 EXTRA POINTS FOR FIVE-OF-A-KIND *)
          ELSE
            SUM := 0        (* NO FIVE-OF-A-KIND *)
   END;
   SCORE[PLAYER][HOW] := SUM;
   WRITELN(TABLES[HOW]);
   WRITELN('FOR A TOTAL OF ',SUM);
   WRITELN
END;

(*
 * PRINTDICE
 * PRINTS OUT THE DICE IN A READABLE FORMAT
 *)
```

```
   PROCEDURE PRINTDICE;
   VAR
      I : DIEINDEX;
   BEGIN
      WRITELN('YOUR DICE LOOK LIKE:');
      FOR I := 1 TO NUMDICE DO
         WRITELN('DIE #-',I,' ',DICE[I].VALUE:2,' ',COLORS[DICE[I].COLOR]);
      WRITELN
   END;

   (*
    * REPLACE
    * WILL ASK FOR A NUMBER (NUM), THEN WILL
    * REPLACE NUM DICE
    *)
   PROCEDURE REPLACE;
   VAR
      NUM : INTEGER;           (* NUMBER INPUT FROM USER *)
      USED : SET OF 1..NUMDICE;
      NUMREP : 1..NUMDICE;     (* NUMBER TO REPLACE *)
   BEGIN
      USED := [];
      REPEAT
         WRITE('REPLACE HOW MANY DICE? ');
         READLN(NUM)
      UNTIL (NUM > 0 ) AND ( NUM <= NUMDICE);

      (*
       * CYCLE THROUGH NUM TIMES REPLACING 1 DIE EACH TIME
       *)
      FOR NUMREP := NUM DOWNTO 1 DO
      BEGIN
         REPEAT
            REPEAT
               WRITE('REPLACE WHICH DIE? ');
               READLN(NUM)
            UNTIL (NUM > 0 ) AND (NUM <= NUMDICE)
         UNTIL NOT ( NUM IN USED);
         USED := USED + [NUM];
         ROLLDIE(DICE[NUM])
      END;
      ROLL := ROLL +1
   END;

(* PLAY *)
BEGIN
   WRITELN('PLAYER NUMBER ',PLAYER);
   FOR DIENUM := 1 TO NUMDICE DO
      ROLLDIE(DICE[DIENUM]);
   ROLL := 1;
   WHILE ROLL < 4 DO
   BEGIN
      REPEAT
         WRITELN;
         PRINTDICE;
         WRITELN('P(RINT,E(VERYONE,S(CORE,R(EPLACE ');
         READLN(CH)
      UNTIL (CH IN ['P','E','S','R']);
      CASE CH OF
         'P': PRINT(PLAYER);

         'E': FOR I := 1 TO NUMPLAYERS DO
                 PRINT(I);

         'S': ROLL := 4;               (* FORCE IT TO QUIT *)

         'R': REPLACE
      END
   END;
   PRINTDICE;
   REPEAT
      REPEAT
         WRITE('SCORING NUMBER? ');
         READLN(NUM)
      UNTIL ( NUM > 0 ) AND ( NUM <= SCORES);
   UNTIL SCORE[PLAYER][NUM] = 0;
   POINTS(NUM)
END;

(*
 * INIT
 * INIT STRINGS
 *)
PROCEDURE INIT;
VAR
   I : DIECOLOR;
   J : TABLETYPE;
   K : 1..MAXPLAYERS;

BEGIN
   FOR K := 1 TO NUMPLAYERS DO
      FOR J := 1 TO SCORES DO
         SCORE[K][J] := 0;
   TABLES[1] := '1 - ACES    1 FOR EACH ACE               ';
   TABLES[2] := '2 - DEUCES  2 FOR EACH DEUCE             ';
   TABLES[3] := '3 - TREYS   3 FOR EACH TREY              ';
   TABLES[4] := '4 - FOURS   4 FOR EACH FOUR              ';
   TABLES[5] := '5 - FIVES   5 FOR EACH FIVE              ';
   TABLES[6] := '6 - SIXES   6 FOR EACH SIX               ';
   TABLES[7] := '7 - 2 PAIR  SAME COLOR   TOTAL DICE      ';
   TABLES[8] := '8 - 3 OF A KIND          TOTAL DICE      ';
   TABLES[9] := '9 - STRAIGHT             30 POINTS       ';
   TABLES[10]:= '10- FLUSH   SAME COLOR   35 POINTS       ';
   TABLES[11]:= '11- FULL HOUSE           TOTAL DICE +15  ';
   TABLES[12]:= '12- FULL HOUSE SAME COLOR TOTAL DICE +20 ';
   TABLES[13]:= '13- 4 OF A KIND          TOTAL DICE +25  ';
   TABLES[14]:= '14- YARBOROUGH  FREE TURN TOTAL DICE     ';
   TABLES[15]:= '15- KISMET  5 OF A KIND  TOTAL DICE +50  ';
   COLORS[RED]:= ' RED ';
   COLORS[GREEN]:= ' GREEN ';
   COLORS[BLACK]:= ' BLACK '
END;
```

```
(* MAIN PROGRAM *)
BEGIN
   REPEAT
      WRITE('HOW MANY PLAYERS? ');
      READLN(NUMPLAYERS)
   UNTIL (NUMPLAYERS > 0) AND (NUMPLAYERS <= MAXPLAYERS);
   RANDOMIZE;                   (* GET A UNIQUE RANDOM NUMBER *)
   INIT;
   FOR CURGAME  := 1 TO MAXPLAYS  DO
      FOR J := 1 TO NUMPLAYERS DO
         PLAY(J);
   (*
    * NOW THAT IT'S ALL DONE,
    * PRINT OUT THE RESULTS
    *)
   FOR J := 1 TO NUMPLAYERS DO
      PRINT(J)
END.
```

# NUMCONVERT

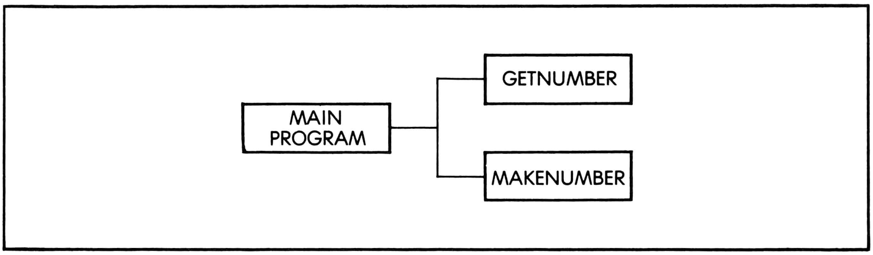
**The Game.** Here is a little exercise for all computer buffs who enjoy trying to beat the computer at its own game. It lets you practice converting numbers into and out of base 2 (binary), base 8 (octal), and base 10 (decimal). Each round consists of ten conversion exercises; at the beginning of a round you are given your choice of conversions:

1. five binary-to-decimal conversions and five decimal-to-binary conversions
2. five binary-to-octal conversions and five octal-to-binary conversions
3. five octal-to-decimal conversions and five decimal-to-octal conversions.

The program displays the number to be converted, accepts your answer, lets you know if you are wrong, and then moves on to the next exercise. At the end of the ten exercises you will be told how many you got right, and given a chance for another round.

```
WHICH WOULD YOU LIKE?
1) BINARY TO DECIMAL
2) BINARY TO OCTAL
3) OCTAL  TO DECIMAL
WHICH NUMBER? 1
BINARY: 00101 ->  DECIMAL: 5
BINARY: 10110 ->  DECIMAL: 22
BINARY: 11101 ->  DECIMAL: 29
BINARY: 10100 ->  DECIMAL: 20
BINARY: 00100 ->  DECIMAL: 4
DECIMAL: 87 ->  BINARY: 1010111
DECIMAL: 73 ->  BINARY: 1000111
WRONG!!
DECIMAL: 53 ->  BINARY: 110101
DECIMAL: 10 ->  BINARY: 1010
DECIMAL: 38 ->  BINARY: 100110

YOU GOT  9 OUT OF 10

AGAIN? N
```

# The Program.



As we will see, program NUMCNVRT contains several rather fascinating features that are worthy of careful attention; these include:

- an implementation of a simple, elegant algorithm to convert a number from one base to another
- a good example of how and when to use the Pascal ORD function
- a function call that contains another function call as one of its parameters
- examples of the "dynamic" STRING type, and of the UCSD LENGTH function.

At the beginning of the main program, the options are displayed, and the user inputs a 1, 2, or 3 to choose an exercise. The input is validated with the GET procedure and INPUT^. INPUT^ is also used as the **CASE** "selector" to assign the correct values (BINARY, OCTAL, or DECIMAL) to the variables FROMBASE and TOBASE according to the exercise option the user chooses. These two variables are declared to be of the scalar type BASE in the **VAR** declaration section. BASE is defined in the **TYPE** declaration section:

```
BASE = (BINARY, OCTAL, DECIMAL)
```

The rounds of the game itself are controlled by two **FOR** loops near the end of the main program. Each loop repeats the execution of one **IF** statement five times. This **IF** statement contains the calls to both functions—MAKENUMBER and GETNUMBER. MAKENUMBER chooses a random number (in the base indicated by the variable FROMBASE), displays it to the player, and calculates and returns the base-10 value of that number to the main program. GETNUMBER requests the player's input for the conversion exercise, converts the input to base 10, and compares the result to the base-10 value of the original number chosen in MAKENUMBER. GETNUMBER thus returns a BOOLEAN value indicating whether the player's conversion is right or wrong. The **FOR** loops in the main program increment the variable RIGHT by 1 for every correct answer.

Notice how economically these **FOR** loops call the functions and test the results. Several statements are actually implied:

```
FOR I := 1 TO 5 DO
  BEGIN
    NUM := MAKENUMBER(FROMBASE);
    CORRECT := GETNUMBER(TOBASE, NUM);
    IF CORRECT THEN
      RIGHT := RIGHT + 1
  END;
```

But these statements are combined in one, eliminating the need for variables NUM and CORRECT:

```
FOR I := 1 TO 5 DO
  IF GETNUMBER(TOBASE, MAKENUMBER(FROMBASE))
    THEN RIGHT := RIGHT + 1;
```

In the second **FOR** loop, FROMBASE and TOBASE are interchanged. For example, if the first five conversions were binary to octal, the next five will be octal to binary.

In function MAKENUMBER, the variable NUM is used in a digit-by-digit calculation of the base-10 value of a number. NUM is initialized to zero at the beginning of the function. Following this initialization, a **CASE** statement selects the appropriate algorithm for producing a random number and converting it to base 10. The **CASE** "selector" is FIRST, the formal parameter of MAKENUMBER that receives a value of type BASE (specifically, the base of the number that is to be converted) from the main program.

The structure of the three **CASE** algorithms is the same. Within a **FOR** loop each digit of the number is chosen randomly, the digit-by-digit conversion to base 10 is performed, and each digit is displayed for the player. For BINARY, five digits are chosen; for OCTAL and DECIMAL, two digits are chosen. The constants BASE2, BASE8, and BASE10, defined in the **CONST** declaration section, are used in combination with the RANDOM function and the **MOD** operation to choose each digit. For example,

```
I := RANDOM MOD BASE2;
```

produces either a 0 or a 1. Then,

```
NUM := NUM * BASE2 + I;
```

performs iterative multiplications by 2 on each digit of the number, from left to right. Thus, the first digit is multiplied by $2^4$, the second by $2^3$, and so on, to compute the base-10 value of the number. Finally,

```
WRITE(I:1)
```

displays each digit of the number at the end of a loop iteration.

At the end of the function, MAKENUMBER is assigned the value of NUM, to be returned to the main program.

Function GETNUMBER has two formal parameters: READBASE receives a value of type BASE (specifically, the base into which the number is to be converted); and ANSWER receives the base-10 value of the number. READBASE is used as the selector in a **CASE** statement that displays the conversion type to the player and assigns the appropriate value to the variable FACTOR. The player's try at converting the number is read into the STRING variable CH, and the length (number of elements) of CH is assigned to the variable STRLENGTH, through use of the UCSD function LENGTH:

```
STRLENGTH := LENGTH(CH);
```

GETNUMBER uses a variation of the same algorithm used in MAKENUMBER to convert the player's calculation (estimate, guess?) into base 10:

```
INTER := INTER * FACTOR + (ORD(CH[INDEX]) - ORD('0'));
```

The only real difference (aside from the different variable names) is the method of referencing each succeeding digit of the number. CH[INDEX] refers to an element of a STRING, which is converted into the corresponding numeric code by use of the Pascal function ORD. Subtracting ORD('0') produces the desired digit from 0 to 9.

Each digit of the player's input is validated before it is calculated into INTER:

```
IF CH[INDEX] IN ['0'..'9'] THEN . . .
```

and if the digit is not numeric, the **WHILE** loop that controls this algorithm is forced to end:

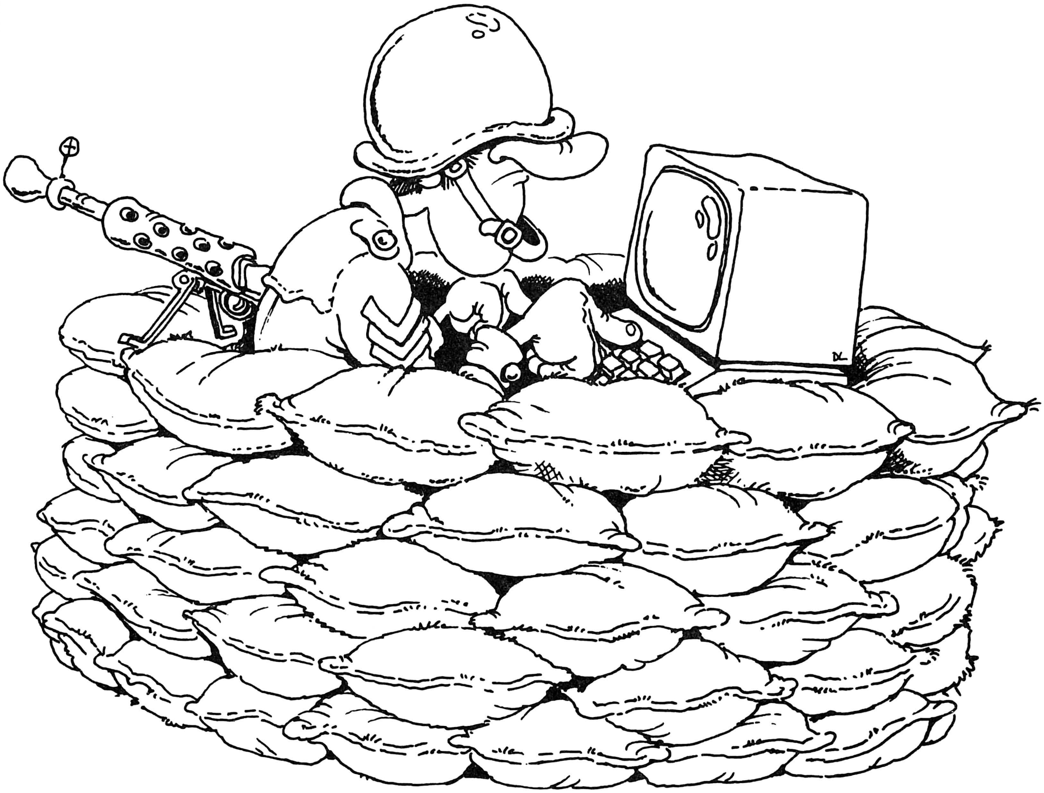
```
ELSE
  INDEX := STRLENGTH;
```

GETNUMBER returns a TRUE or FALSE value to the main program, depending on whether the player's answer is correct or incorrect.

Finally, after the player has attempted the ten conversions, the main program displays the number right out of 10 and offers the player another round.

```
(*
 * NUMCNVRT
 * TESTS YOUR ABILITY TO CONVERT A NUMBER
 * FROM BASES
 * BINARY -> DECIMAL, DECIMAL -> BINARY
 * DECIMAL -> OCTAL, OCTAL -> DECIMAL
 * BINARY -> OCTAL, OCTAL -> BINARY
 *)
PROGRAM NUMCNVRT(INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   BASE2        =  2;           (* BINARY *)
   BASE8        =  8;           (* OCTAL *)
   BASE10       = 10;           (* DECIMAL *)
TYPE
   BASE = (BINARY,OCTAL,DECIMAL);
VAR
   I   : INTEGER;               (* INDEX VARIABLE *)
   RIGHT:INTEGER;               (* NUMBER OF CORRECT GUESSES *)
   CH   :CHAR;                  (* USED TO ANSWER QUESTIONS *)
   FROMBASE: BASE;              (* FROM THIS BASE *)
   TOBASE  : BASE;              (* TO THIS BASE *)


   (*
    * MAKENUMBER
    * CREATES A RANDOM NUMBER OF BASE
    * FIRST, THEN PRINTS IT OUT IN BASE
    * FIRST. IT THEN RETURNS A DECIMAL REPRESENTATION
    * OF THAT NUMBER
    *)
   FUNCTION MAKENUMBER(FIRST:BASE):INTEGER;
   VAR
      TEMP : STRING;
      NUM  : INTEGER;
      INDEX: INTEGER;
      I    : INTEGER;
   BEGIN
      NUM := 0;
      CASE FIRST OF
         BINARY: BEGIN
                   WRITE('BINARY: ');FOR INDEX := 1 TO 5 DO
                   BEGIN
                      I := RANDOM MOD BASE2;
                      NUM := NUM * BASE2 + I;
                      WRITE(I:1)
                   END
                END;

         OCTAL: BEGIN
                  WRITE('OCTAL: ');
                  FOR INDEX := 1 TO 2 DO
                  BEGIN
                     I := RANDOM MOD BASE8;
                     NUM := NUM * BASE8 + I;
                     WRITE(I:1)
                  END
               END;
      DECIMAL:BEGIN
                 WRITE('DECIMAL: ');
                 FOR INDEX := 1 TO 2 DO
                 BEGIN
                    I := RANDOM MOD BASE10;
                    NUM := NUM*BASE10 + I;
                    WRITE(I:1)
                 END
              END
   END;
   MAKENUMBER := NUM
END; (* MAKENUMBER *)

(*
 * GETNUMBER
 * GETS A NUMBER FROM THE USER, IN THE
 * PASSED BASE. IT THEN CHECKS TO SEE
 * IF IT IS EQUAL TO THE DECIMAL ANSWER
 * IT WAS PASSED. IT RETURNS A BOOLEAN INDICATING
 * WHETHER OR NOT THEY MATCHED
 *)
FUNCTION GETNUMBER(READBASE : BASE; ANSWER:INTEGER):BOOLEAN;
VAR
   CH  :  STRING;             (* INPUT FROM USER *)
   FACTOR:INTEGER;            (* MULTIPLYING FACTOR *)
   INTER :INTEGER;            (* INTERMEDIATE RESULTS *)
   INDEX :INTEGER;            (* INDEX INTO CH *)
   STRLENGTH:INTEGER;         (* LENGTH OF READ IN STRING *)

BEGIN
   WRITE(' -> ');
   CASE READBASE OF
      DECIMAL: BEGIN
                  WRITE(' DECIMAL: ');
                  FACTOR := BASE10
               END;
      OCTAL   : BEGIN
                  WRITE(' OCTAL: ');
                  FACTOR := BASE8
               END;
      BINARY : BEGIN
                  WRITE(' BINARY: ');
                  FACTOR := BASE2
               END
   END;
   READLN(CH);
   STRLENGTH := LENGTH(CH);
   INDEX := 1;
   INTER := 0;
   WHILE (STRLENGTH >= INDEX) DO
   BEGIN
      IF CH[INDEX] IN ['0'..'9'] THEN
         INTER := (INTER * FACTOR) + (ORD(CH[INDEX]) - ORD('0'))
      ELSE
         INDEX := STRLENGTH;
      INDEX := INDEX +1
   END;
```

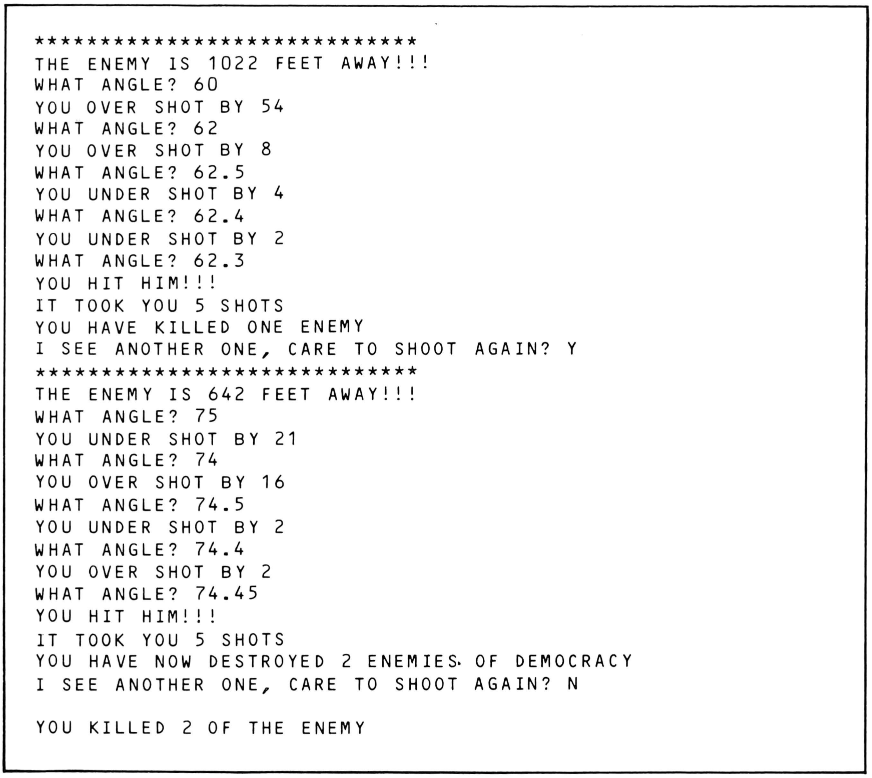
```
    IF INTER = ANSWER THEN
      GETNUMBER := TRUE
    ELSE
    BEGIN
      WRITELN('WRONG!!');
      GETNUMBER := FALSE
    END
  END; (* GETNUMBER *)
BEGIN
  WRITELN('WHICH WOULD YOU LIKE?');
  WRITELN('1) BINARY TO DECIMAL');
  WRITELN('2) BINARY TO OCTAL');
  WRITELN('3) OCTAL  TO DECIMAL');

  REPEAT
    WRITE('WHICH NUMBER? ');
    GET(INPUT);
    WHILE NOT (INPUT^ IN ['1'..'3']) DO
    BEGIN
      READLN(INPUT);
      WRITELN('ILLEGAL NUMBER');
      WRITE('WHICH NUMBER? ');
      GET(INPUT)
    END;
    CASE INPUT^ OF
      '1' : BEGIN
              FROMBASE := BINARY;
              TOBASE   := DECIMAL
            END;
      '2' : BEGIN
              FROMBASE := BINARY;
              TOBASE   := OCTAL
            END;
      '3' : BEGIN
              FROMBASE := OCTAL;
              TOBASE   := DECIMAL
            END
    END;
    READLN(INPUT);
    RANDOMIZE;
    RIGHT := 0;
    FOR I := 1 TO 5 DO
      IF GETNUMBER(TOBASE ,MAKENUMBER(FROMBASE)) THEN
        RIGHT := RIGHT +1;

    (* NOW REVERSE THE BASES *)
    FOR I := 1 TO 5 DO
      IF GETNUMBER(FROMBASE,MAKENUMBER(TOBASE)) THEN
        RIGHT := RIGHT +1;

    WRITELN;
    WRITELN('YOU GOT ',RIGHT:2,' OUT OF 10');
    WRITELN;
    WRITE('AGAIN? ');
    READLN(CH)
  UNTIL CH <> 'Y'
END. (* NUMCNVRT *)
```



# ARTILLERY

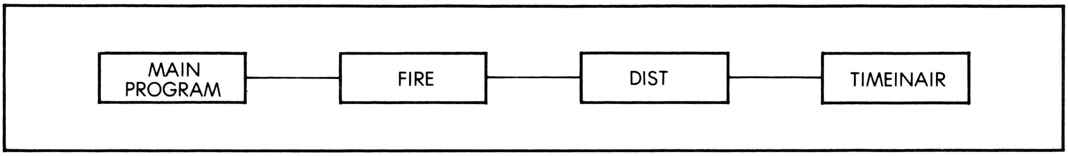
**The Game.** This war game is of the same genre as the games you may have paid a quarter to play at your local bar or pinball palace, the main difference being that here the game is in words rather than in graphics. The object is to hit the "enemy" target with a cannon shell. You are given the location of the target (i.e., its distance from the cannon), and you must decide on the shooting angle (the angle of projection). The initial velocity of the shell is 200 ft/sec. The program computes the trajectory of the shell and tells you the distance between the hit and the target. For any given target you have a maximum of ten shots, and the program continues to display targets as long as you want to play.

```
WELCOME TO ARTILLERY

YOU ARE IN THE MIDDLE OF A WAR (DEPRESSING, NO?)
AND ARE BEING CHARGED BY THOUSANDS
OF ENEMIES.
YOUR JOB IS TO DESTROY THEIR OUTPOSTS.
YOU HAVE AT YOUR DISPOSAL,
A CANNON, WHICH YOU CAN SHOOT AT ANY ANGLE.
AS THIS IS WAR, SUPPLIES ARE SHORT, SO
YOU ONLY HAVE 10 PER TARGET
```



```
*****************************
THE ENEMY IS 1022 FEET AWAY!!!
WHAT ANGLE? 60
YOU OVER SHOT BY 54
WHAT ANGLE? 62
YOU OVER SHOT BY 8
WHAT ANGLE? 62.5
YOU UNDER SHOT BY 4
WHAT ANGLE? 62.4
YOU UNDER SHOT BY 2
WHAT ANGLE? 62.3
YOU HIT HIM!!!
IT TOOK YOU 5 SHOTS
YOU HAVE KILLED ONE ENEMY
I SEE ANOTHER ONE, CARE TO SHOOT AGAIN? Y
*****************************
THE ENEMY IS 642 FEET AWAY!!!
WHAT ANGLE? 75
YOU UNDER SHOT BY 21
WHAT ANGLE? 74
YOU OVER SHOT BY 16
WHAT ANGLE? 74.5
YOU UNDER SHOT BY 2
WHAT ANGLE? 74.4
YOU OVER SHOT BY 2
WHAT ANGLE? 74.45
YOU HIT HIM!!!
IT TOOK YOU 5 SHOTS
YOU HAVE NOW DESTROYED 2 ENEMIES OF DEMOCRACY
I SEE ANOTHER ONE, CARE TO SHOOT AGAIN? N

YOU KILLED 2 OF THE ENEMY
```

# The Program.



The main task of this program is to compute the range of a trajectory, given the initial velocity and the angle of projection. The mathematical algorithms for this task are contained in functions DIST and TIMEINAIR. The main program merely sets the scene with a series of WRITELN statements and makes repeated calls to procedure FIRE as long as the player continues to request more targets.

Procedure FIRE begins by using the RANDOM function to determine the position of the enemy:

```
ENEMY := MINDIST + RANDOM MOD MAXDIST;
```

FIRE then prompts the player for an angle, converts the angle from degrees into radians and calls function DIST to compute the distance.

If the value returned by DIST is within a range of one unit from the enemy position—that is, if the BOOLEAN variable HIT is assigned a TRUE value:

```
HIT := ABS(ENEMY - DIST) <= 1;
```

then several congratulatory messages are printed. Otherwise the undershot or overshot distance is computed and displayed:

```
WRITELN (ABS(ENEMY - DIST))
```

All this is repeated (in a **REPEAT/UNTIL** loop) until the ten allotted shells are used up (SHOTS = 0) or the enemy is hit.

The algorithms of functions DIST and TIMEINAIR come from the equations for the x and $y$ components of a trajectory:

$$x = v_0 t \cos\theta \qquad (1)$$
$$y = v_0 t \sin\theta - \tfrac{1}{2} g t^2 \qquad (2)$$

where $v_0$ is the initial velocity of the projectile, $\theta$ is the angle of projection, $t$ is the time, and g is the acceleration due to gravity (32.2 ft/sec$^2$).

To find the amount of time the projectile is in the air, we set Equation 2 equal to zero and solve for $t$:

$$t = 0 \qquad (3)$$

and

$$t = \frac{2v_0 \sin\theta}{g} \qquad (4)$$

Equation 4 is the algorithm used in function TIMEINAIR:

```
TIMEINAIR := (2 * VELOCITY * SIN(ANGLE))/GRAVITY
```

Now, to find the "range" (or distance travelled) of the projectile, we substitute Equation 4 into Equation 1:

$$x = v_0 \cos\theta \frac{2v_0 \sin\theta}{g} \tag{5}$$

or

$$x = \frac{2v_0^2 \sin\theta \cos\theta}{g} \tag{6}$$

Equation 5 is the algorithm of function DIST:

```
DIST := ROUND((VELOCITY * COS(ANGLE)) * TIMEINAIR)
```

If you find yourself getting involved in the mathematics of this game, you might want to add a curve-plotting routine to the program to plot the curve of the projectile.[1] This would enhance the ARTILLERY game with some graphical output.

---

[1] For a discussion of curve plotting in Pascal, see *Pascal Programs for Scientists and Engineers*, by Alan R. Miller (SYBEX, 1981).

```
(*
 * ARTILLERY
 * FIRE A SHELL AT AN ENEMY OUTPOST
 *)
PROGRAM ARTILLERY(INPUT,OUTPUT);
USES APPLESTUFF, TRANSCEND;
CONST
   NUMSHELLS    =  10;          (* ALLOWED 10 SHELLS PER TARGET *)
   MINDIST      = 100;          (* MINIMUM DISTANCE FOR A TARGET *)
   MAXDIST      =1000;          (* MAXIMUM    "      "  "    "   *)

   VELOCITY     = 200.0;        (* INITIAL VELOCITY OF 200 FT/SEC *)
   GRAVITY      = 32.2;         (* GRAVITY OF 32.2 FT/SEC^2 *)
   PI           = 3.1415;
VAR
   ANGLE : REAL;                (* ANGLE TO SHOOT AT *)
   ENEMY : INTEGER;             (* HOW FAR AWAY THE ENEMY IS *)
   KILLED: INTEGER;             (* HOW MANY WE HAVE HIT *)
   SHOTS : 0..NUMSHELLS;        (* NUMBER OF SHELLS LEFT *)
   CH    : CHAR;                (* USED TO ANSWER QUESTIONS *)
   HIT   : BOOLEAN;             (* WHETHER THE ENEMY HAS BEEN HIT *)

   (*
    * DIST
    * RETURNS HOW FAR THE SHELL WENT
    *)
   FUNCTION DIST:INTEGER;

      (*
       * TIMEINAIR
       * FIGURES OUT HOW LONG THE SHELL
       * STAYS IN THE AIR
       *)
      FUNCTION TIMEINAIR:REAL;
      BEGIN
         TIMEINAIR := (2*VELOCITY * SIN (ANGLE))/ GRAVITY
      END; (* TIMEINAIR *)

   BEGIN
      DIST := ROUND( (VELOCITY * COS(ANGLE)) * TIMEINAIR )
   END; (* DIST *)

   (*
    * FIRE
    * THE USER FIRES AT THE ENEMY
    *)
   PROCEDURE FIRE;
   BEGIN
      RANDOMIZE;
      ENEMY := MINDIST + RANDOM MOD MAXDIST;

      WRITELN('THE ENEMY IS ',ENEMY:3,' FEET AWAY!!!');
      SHOTS := NUMSHELLS;
      REPEAT
         WRITE('WHAT ANGLE? ');
         READLN(ANGLE);
         ANGLE := (ANGLE * PI)/180.0;   (* CONVERT FROM DEGREES TO RADIANS *)
         HIT := ABS(ENEMY - DIST) <= 1;
         IF HIT THEN
         BEGIN
            KILLED := KILLED +1;
            WRITELN('YOU HIT HIM!!!');
            WRITELN('IT TOOK YOU ',(NUMSHELLS-SHOTS + 1),' SHOTS');
            IF KILLED = 1 THEN
              WRITELN('YOU HAVE KILLED ONE ENEMY')
            ELSE
              WRITELN('YOU HAVE NOW DESTROYED ',KILLED,' ENEMIES OF DEMOCRACY')
         END
         ELSE
         BEGIN
            SHOTS := SHOTS -1;
            IF DIST > ENEMY THEN
               WRITE('YOU OVER SHOT BY ')
            ELSE
               WRITE('YOU UNDER SHOT BY ');
            WRITELN(ABS(ENEMY - DIST))
         END
      UNTIL (SHOTS = 0) OR HIT;

      IF SHOTS = 0 THEN
         WRITELN('YOU HAVE RUN OUT OF AMMO')
   END; (* FIRE *)


BEGIN
   WRITELN('WELCOME TO ARTILLERY');
   WRITELN;
   WRITELN('YOU ARE IN THE MIDDLE OF A WAR (DEPRESSING, NO?)');
   WRITELN('AND ARE BEING CHARGED BY THOUSANDS');
   WRITELN('OF ENEMIES.');
   WRITELN('YOUR JOB IS TO DESTROY THEIR OUTPOSTS.');
   WRITELN('YOU HAVE AT YOUR DISPOSAL, ');
   WRITELN('A CANNON, WHICH YOU CAN SHOOT AT ANY ANGLE.');
   WRITELN('AS THIS IS WAR, SUPPLIES ARE SHORT, SO');
   WRITELN('YOU ONLY HAVE ',NUMSHELLS,' PER TARGET');
   WRITELN;

   KILLED := 0;
   REPEAT
      WRITELN('*****************************');
      FIRE;
      WRITE('I SEE ANOTHER ONE, CARE TO SHOOT AGAIN? ');
      READLN(CH);
```

```
    WHILE NOT (CH IN ['Y','N']) DO
    BEGIN
      WRITELN('PLEASE ANSWER YES OR NO');
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
    END
  UNTIL CH <> 'Y';

  WRITELN;
  WRITELN('YOU KILLED ',KILLED,' OF THE ENEMY')
END. (* ARTILLERY 7/7/81 D.H. *)
```

# PASCAL

*L'homme n'est qu'un roseau, le plus faible*
*de la nature; mais c'est un roseau pensant.*
— Pascal, *Pensées*, vi, 347

**The Game.**[1] It seems especially fitting that a program designed to create Pascal triangles (and then to make lovely graphical designs out of them) should be written in Pascal. The language, after all, was so named as a tribute to the seventeenth century French mathematician and philosopher. More than that, however, it turns out that the most elegant way to create the triangles (in a computer) is via an essential Pascal data structure—the dynamic record variable. But we are getting slightly ahead of our story.

A Pascal triangle is a triangular arrangement of integers in which each entry is equal to the sum of the two entries directly above it. Here, for example, is a Pascal triangle with nine rows:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

---

[1] Two BASIC versions of this game were written by Charles A. Lund, and appear in *BASIC Computer Games* edited by David H. Ahl (Morristown, N.J.: Creative Computing, 1980).

Notice, first, that each side of the triangle is made up of a diagonal of 1s. Also notice that the triangle is symmetrical around the line of numbers that represents the "height" of the triangle (1, 2, 6, 20, 70, . . . ). Part of the mathematical significance of this triangle is that it provides the *coefficients* of a binomial expansion. For example:

$(x + a)^0 = 1$ (first row)
$(x + a)^1 = x + a$ (second row)
$(x + a)^2 = x^2 + 2ax + a^2$ (third row)
$(x + a)^3 = x^3 + 3a^2x + 3ax^2 + a^3$ (fourth row)

. . . and so on.

This program asks you two questions before creating a triangle:

```
WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES?
```

and

```
HOW BIG IS THE TRIANGLE?
```

Begin by answering the first question with a 0 and the second question with the number of rows you want to see, say 9. Notice that the program rotates the triangle:

```
1   1   1   1   1   1   1   1   1
1   2   3   4   5   6   7   8
1   3   6   10  15  21  28
1   4   10  20  35  56
1   5   15  35  70
1   6   21  56
1   7   28
1   8
1
```

Now each number is the sum of the number directly above it and the number directly to its left.

Next, you will see the question:

```
CONTINUE?
```

Answer with a Y if you wish to see another triangle. (Any other answer will terminate the program.) The two original questions will be displayed once again. This time, answer the first question with a nonzero positive integer, say 5. Enter a 9 again for the number of rows. What you will discover is that any nonzero answer to the first question produces not a Pascal triangle, but a graphics design based on the triangle. By entering a 5, you are asking the program to replace all the numbers of the triangle that are evenly divisible by 5 with a space. The rest of the numbers (i.e., those that are *not* evenly divisible by 5) will be replaced by an asterisk character, producing a design something like this:

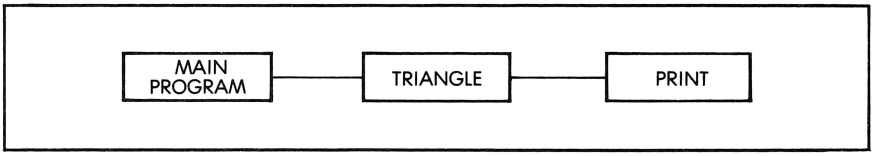
```
*   *   *   *   *   *   *   *   *
*   *   *   *       *   *   *
*   *   *           *   *
*   *               *
*
*   *   *   *
*   *   *
*   *
*
```

Experiment with this program to see what different kinds of designs you can produce. At some point, try entering a 1 as the multiple. (Of course, *all* of the numbers of the triangle are divisible by 1.) If you watch closely you will witness a truly mystical phenomenon: the creation of an *invisible* Pascal triangle!

```
WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES? 0
SIZE OF THE TRIANGLE? 10
   1   1   1   1   1   1   1   1   1   1
   1   2   3   4   5   6   7   8   9
   1   3   6  10  15  21  28  36
   1   4  10  20  35  56  84
   1   5  15  35  70 126
   1   6  21  56 126
   1   7  28  84
   1   8  36
   1   9
   1


CONTINUE? Y
WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES? 2
SIZE OF THE TRIANGLE? 15
* * * * * * * * * * * * * * *
*   *   *   *   *   *   *
* *     * *     * *     *
*       *       *
* * * *         * * *
*   *           *
* *             *
*
* * * * * * *
*   *   *
* *     *
*
* * *
*
*


CONTINUE? Y
WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES? 1
SIZE OF THE TRIANGLE? 15















CONTINUE? Y
```

```
WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES? 4
SIZE OF THE TRIANGLE? 30
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * *   * * *   * * *   * * *   * * *   * * *   * * *   *
* * * * * *     * * * * * *     * * * * * *     * * * *
*   *   *       *   *   *       *   *   *       *   *
* * * * * * * * * * * *         * * * * * * * * * *
* * *   *   *   * * *           * * *   *   *   *
* *     * *     * *             * *     * *
*       *       *               *       *
* * * * * * * * * * * * * * * * * * * * * *
* * *   * * *   *   *   *   *   * * *   *
* * * * * *     * *     * *     * * * *
*   *   *       *       *       *   *
* * * *         * * * *         * *
* * *           *   *           *
* *             * *
*               *
* * * * * * * * * * * * * *
* * *   * * *   * * *   *
* * * * * *     * * * *
*   *   *       *   *
* * * * * * * * * *
* * *   *   *   *
* *     * *
*       *
* * * * * *
* * *   *
* * * *
*   *
* *
*

CONTINUE? N
```

# The Program.[2]



This short program provides a good introduction to the use of dynamic variables in Pascal. A dynamic record type is used to create a "linked list." The flexibility of this list lies in the fact that its length is not defined in advance; the number of records actually created by the program depends on the number that is entered in response to the question:

---

[2] For a clear description of the use of dynamic variables in Pascal, see *Introduction to Pascal, Including UCSD Pascal* by Rodnay Zaks (Berkeley, Calif.: Sybex, 1980) (Chapter 13).

### HOW BIG IS THE TRIANGLE?

Begin by studying the **TYPE** declaration at the beginning of procedure TRIANGLE:

```
TYPE
   NODEPTR = ^NODE;
   NODE = RECORD
      VALUE : INTEGER;
      NEXT : NODEPTR
   END;
```

The first declaration reads, "Type NODEPTR is a pointer to a NODE." (Note that this is the one instance when Pascal allows reference to a data type before that type has been defined.) NODE is then declared as a record type with two fields. VALUE, of type INTEGER, will hold the actual values of the Pascal triangle. NEXT, of type NODEPTR, will be a pointer to the next dynamic record in the linked list.

The main program repeatedly calls procedure TRIANGLE as long as the user responds with a Y to the prompt. Notice the first and the last statements in TRIANGLE:

```
MARK(HEAP);
```

and

```
RELEASE(HEAP)
```

MARK and RELEASE are UCSD procedures, both of which take one parameter, a pointer variable. In this program, HEAP is defined in the variable declarations as:

```
HEAP : ^INTEGER
```

The "heap" is the part of memory where the new dynamic variables are to be stored. The statement MARK(HEAP) stores the first free address of the heap in the pointer variable HEAP. Any new dynamic variables are stored after this address. When those dynamic variables are no longer needed in the program, the statement RELEASE(HEAP) frees the memory space by restoring the top of the heap to its initial address. (Note that any variable name could have been used with MARK and RELEASE—there is nothing special about the name HEAP. However, once an address is assigned to a pointer variable via a MARK statement, the value of that variable should not be changed until a RELEASE is executed.)

The first half of procedure TRIANGLE prompts the reader to input values for MULT and SIZE. Both values are validated via a GET statement before a READLN is executed.

The next **FOR** loop controls the creation of the dynamic variables. The number of variables to be created is equal to the value of SIZE. The first statement of the loop calls PRINT:

```
PRINT(1);
```

This is to create the first row of 1s (or asterisks) in the Pascal triangle. (As we will see, procedure PRINT writes an integer, an asterisk, or a space, depending on the value of MULT.)

Then a variable of type NODE is created by the statement:

```
NEW(CURNODE);
```

and the fields of this new variable are initialized:

```
CURNODE^.VALUE := 1;
CURNODE^.NEXT := START;
```

START is a pointer variable that always points to the last dynamic variable that was created:

```
START := CURNODE;
```

Thus, the NEXT field of each record is a pointer variable that points to the previously created variable in the "linked list."

The remaining rows of the triangle are created by the nested **FOR** loops that follow. The outer loop:

```
FOR LEFT := 1 TO SIZE DO
```

controls the creation of each row. The inner loop:

```
FOR WHICH := 1 TO (SIZE - LEFT) DO
```

determines the value of each element in a row. (Each row has one element fewer than the row above it.) Note that no new variables are created in these loops; the VALUE fields of the already-created dynamic records are reset to represent the current row.

LAST is initialized to 0, and CURNODE points to the top of the list (CURNODE := START) before each row is created. Thus the statement:

```
PRINT(LAST + CURNODE^.VALUE)
```

will send the value 1 to procedure PRINT at the beginning of each row. In addition, the statement:

```
CURNODE^.VALUE := CURNODE^.VALUE + LAST;
```

will store a 1 in the VALUE field at the beginning of each row. So that each element of the triangle will equal the sum of the elements above it and to its left, LAST is then set equal to the present VALUE:

```
LAST := CURNODE^.VALUE;
```

Finally, the pointer variable CURNODE is "incremented" so that it points to the next variable in the heap:

```
CURNODE := CURNODE^.NEXT
```

All variables are now set for the next execution of the inner loop and the calculation of a new element of the triangle.

Procedure PRINT writes each value of the triangle and contains the algorithm for "blanking out" the desired multiples. First, if the user entered a 0 for MULT, then the actual integral values of the triangle are written:

```
IF MULT = 0 THEN
  WRITE (VALUE:4)
```

Otherwise, a blank character is written for each VALUE that is found to be a multiple of MULT:

```
IF (VALUE MOD MULT) = 0 THEN
  WRITE ('  ')
```

and an asterisk character is written for all other values:

```
ELSE
  WRITE ('* ')
```

```
(*
 * PASCAL
 * CREATES AND DRAWS PASCAL TRIANGLES OF
 * ANY SIZE
 *)
PROGRAM PASCAL(INPUT,OUTPUT);
VAR
   CH   : CHAR;           (* USED TO ANSWER QUESTIONS *)

   (*
    * TRIANGLE
    * CREATES THE TRIANGLE, AND PRINTS IT OUT.
    * NOTE THAT ONLY ONE ROW IS CREATED. TO MAKE
    * A NODE, ALL YOU NEED TO KNOW IS THE VALUE OF
    * THE NODE TO ITS LEFT, AND THE VALUE OF THE NODE
    * DIRECTLY ABOVE IT. AS A RESULT, I ONLY KEEP THE
    * PREVIOUS ROW.
    *)
   PROCEDURE TRIANGLE;
   TYPE
      NODEPTR = ^NODE;
      NODE    = RECORD
         VALUE : INTEGER;       (* VALUE OF THIS NODE *)
         NEXT  : NODEPTR        (* NEXT NODE (MAY BE NIL) *)
      END;
   VAR
      START : NODEPTR;          (* BEGINNING OF THE LAST LINE *)
      CURNODE:NODEPTR;          (* CURRENT NODE *)
      SIZE  : INTEGER;          (* HOW BIG THE TRIANGLE IS TO BE *)
      MULT  : INTEGER;          (* THE MULTIPLES TO BE REMOVED *)
      LAST  : INTEGER;          (* VALUE OF THE LAST NODE *)
      HEAP  : ^INTEGER;         (* POINTER TO THE HEAP *)
      WHICH : INTEGER;          (* WHICH NODE WE ARE PRINTING ON THE LINE*)
      LEFT  : INTEGER;          (* HOW MANY ROWS OF THE TRIANGLE LEFT *)

      (*
       * PRINT
       * PRINTS OUT THE CURRENT NODE
       *)
      PROCEDURE PRINT(VALUE:INTEGER);
      BEGIN
         IF MULT = 0 THEN       (* IF MULT = 0 => USER WANTS REAL TRIANGLE *)
            WRITE(VALUE:4)
         ELSE
            IF (VALUE MOD MULT) = 0 THEN
               WRITE('  ')
            ELSE
               WRITE('* ')
      END; (* PRINT *)

   BEGIN
      MARK(HEAP);               (* USED TO FREE UP SPACE *)

      WRITE('WHICH MULTIPLES DO YOU WANT PRINTED AS SPACES? ');
      GET(INPUT);
      WHILE NOT (INPUT^ IN ['0'..'9']) DO
      BEGIN
         READLN(INPUT);
         WRITELN('SORRY, ONLY NUMBERS ALLOWED');
         WRITE('REMOVE WHICH MULTIPLES? ');
         GET(INPUT)
      END;
      READLN(MULT);
      WRITE('SIZE OF THE TRIANGLE? ');
      GET(INPUT);
      WHILE NOT (INPUT^ IN ['1'..'9'])DO
      BEGIN
         READLN(INPUT);
         WRITELN('PLEASE TRY AGAIN');
         WRITE('HOW BIG IS THE TRIANGLE? ');
         GET(INPUT)
      END;
      READLN(SIZE);

      (*
       * CREATE THE FIRST ROW.
       * NOTE THAT IT IS BEING CREATED FROM
       * THE LEFT SIDE TOWARD THE RIGHT.
       *)
      FOR WHICH := 1 TO SIZE DO
      BEGIN
         PRINT(1);
         NEW(CURNODE);
         CURNODE^.VALUE := 1;          (* THE FIRST ROW HAS ALL 1'S *)
         CURNODE^.NEXT  := START;      (* LINK BACKWARD *)
         START := CURNODE              (* RESET START *)
      END;
      WRITELN;

      FOR LEFT := 1 TO SIZE DO
      BEGIN
         LAST := 0;                    (* FORCES THE FIRST NUMBER TO BE A 1 *)
         CURNODE := START;             (* RESET POINTER *)
         FOR WHICH := 1 TO (SIZE - LEFT) DO
         BEGIN
            PRINT(LAST+CURNODE^.VALUE);
            CURNODE^.VALUE := CURNODE^.VALUE + LAST;(* CHANGE THE UPPER ROW *)
            LAST := CURNODE^.VALUE;
            CURNODE := CURNODE^.NEXT
         END;
         WRITELN
      END;
      RELEASE(HEAP)                    (* RELEASE SPACE *)
   END; (* TRIANGLE *)
BEGIN
   REPEAT
      TRIANGLE;
      WRITELN;
      WRITE('CONTINUE? ');
      READLN(CH)
   UNTIL CH <> 'Y'

END. (* PASCAL *)
```

# BACCARAT

**The Game.** Baccarat and its first-cousin, Chemin-de-fer, are popular gambling games in both Europe and the United States. Usually there are many players around the table—all betting against the bank—but in this version you and the computer play alone. Also, this version is played completely as a game of chance—the only decisions you make are, first, whether to be the banker or the player, and, second, how much money to bet (if you decide to be the player).

You begin the game by deciding on your role:

```
DO YOU WANT TO BE THE BANKER, OR PLAYER?
```

You enter a P or a B. If you decide to be the player, you must place a bet against the bank:

```
BET?
```

The stakes are high; you and the computer both begin the game with $10,000. If you decide to be the banker, the computer (i.e., the player) will display its bet against you:

```
THE BET IS . . .
```

Then the play begins. You play with eight decks of cards. All 416 cards are shuffled together, and the cards are dealt, one at a time and face up, two to the player and two to the banker. (All four cards are displayed.)

The value of each hand is calculated. An ace is worth 1; 10s and face cards are worth zero; other number cards carry their face value. Factors of ten are eliminated from the combined value of the two cards in a hand. (For example, a 7 and an 8 have a value of 5, i.e., 15 minus 10.)

In the end, the winning hand will be the one that is closer to a value of 9. If one of the hands has a value of 8 or 9 (a "natural"), the play ends immediately. Otherwise, one more card may be dealt to the player and possibly also to the banker. Decisions on whether or not to deal this third card are made as follows:

*The Player*

- takes another card for a hand worth 5 or less
- "stands" on a hand worth 6 or 7.

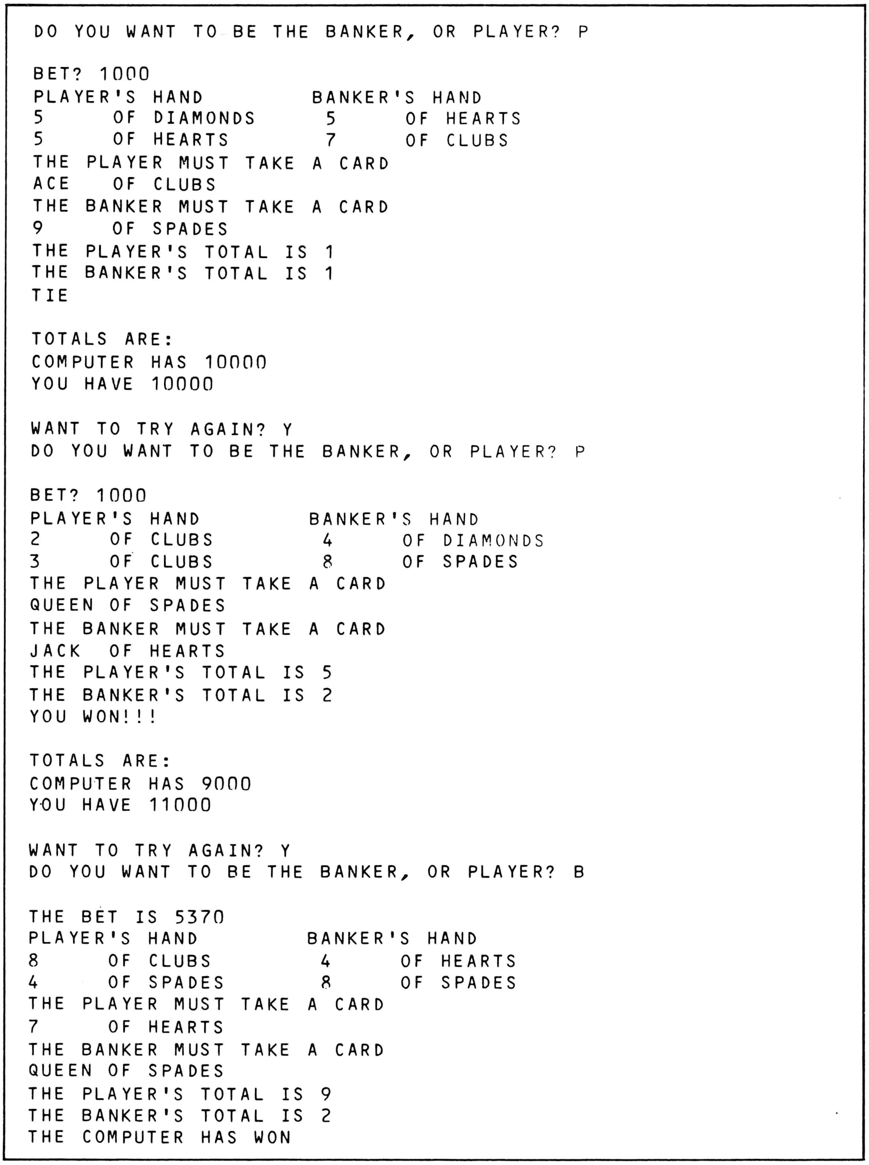
*The Banker*

- if the player "stood" the banker takes another card only if his own hand is worth 5 or less (banker stands on a 6 or 7)
- otherwise, the banker's decision is based on both the value of the third card dealt to the player and on the value of the banker's own hand:

| *If player was dealt a:* | *Banker stands on a hand worth:* | *Banker draws another card if hand is worth:* |
|---|---|---|
| face card, 10, or 9 | 4, 5, 6, or 7 | 3, 2, 1, or 0 |
| 8 | 3, 4, 5, 6, or 7 | 2, 1, or 0 |
| 7, or 6 | 7 | 6, 5, 4, 3, 2, 1, or 0 |
| 5, or 4 | 6, or 7 | 5, 4, 3, 2, 1, or 0 |
| 3, or 2 | 5, 6, or 7 | 4, 3, 2, 1, or 0 |
| Ace | 4, 5, 6, or 7 | 3, 2, 1, or 0 |

After any additional cards are dealt, the value of each resulting hand is calculated, and the winner is announced. The amount of the bet is subtracted from the loser's money and added to the winner's.

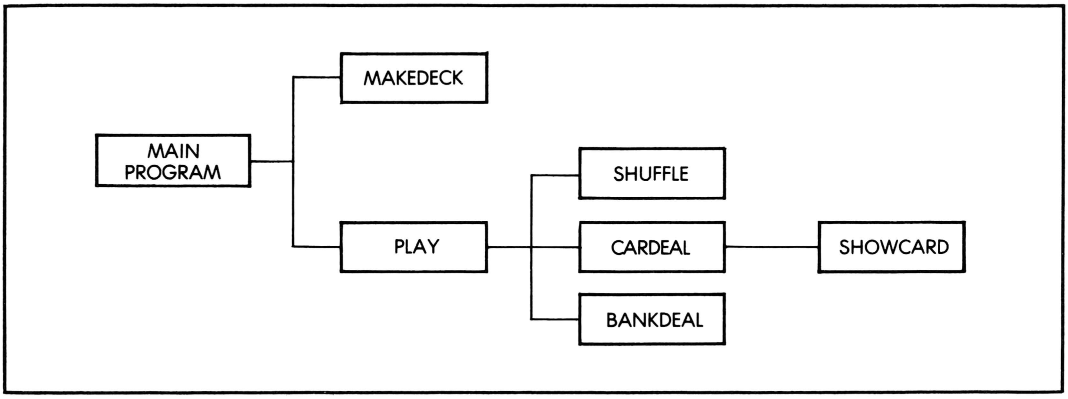
Continue for as many rounds as you like. The house trusts you and will even let you slip into the red; your IOUs are good. This game is best played over *very* dry martinis. No olives for the computer, thank you.



```
DO YOU WANT TO BE THE BANKER, OR PLAYER? P

BET? 1000
PLAYER'S HAND          BANKER'S HAND
5     OF DIAMONDS       5     OF HEARTS
5     OF HEARTS         7     OF CLUBS
THE PLAYER MUST TAKE A CARD
ACE   OF CLUBS
THE BANKER MUST TAKE A CARD
9     OF SPADES
THE PLAYER'S TOTAL IS 1
THE BANKER'S TOTAL IS 1
TIE

TOTALS ARE:
COMPUTER HAS 10000
YOU HAVE 10000

WANT TO TRY AGAIN? Y
DO YOU WANT TO BE THE BANKER, OR PLAYER? P

BET? 1000
PLAYER'S HAND          BANKER'S HAND
2     OF CLUBS          4     OF DIAMONDS
3     OF CLUBS          8     OF SPADES
THE PLAYER MUST TAKE A CARD
QUEEN OF SPADES
THE BANKER MUST TAKE A CARD
JACK  OF HEARTS
THE PLAYER'S TOTAL IS 5
THE BANKER'S TOTAL IS 2
YOU WON!!!

TOTALS ARE:
COMPUTER HAS 9000
YOU HAVE 11000

WANT TO TRY AGAIN? Y
DO YOU WANT TO BE THE BANKER, OR PLAYER? B

THE BET IS 5370
PLAYER'S HAND          BANKER'S HAND
8     OF CLUBS          4     OF HEARTS
4     OF SPADES         8     OF SPADES
THE PLAYER MUST TAKE A CARD
7     OF HEARTS
THE BANKER MUST TAKE A CARD
QUEEN OF SPADES
THE PLAYER'S TOTAL IS 9
THE BANKER'S TOTAL IS 2
THE COMPUTER HAS WON
```

```
TOTALS ARE:
COMPUTER HAS 14370
YOU HAVE 5630

WANT TO TRY AGAIN? Y
DO YOU WANT TO BE THE BANKER, OR PLAYER? B

THE BET IS 410
PLAYER'S HAND        BANKER'S HAND
9     OF CLUBS        9     OF CLUBS
2     OF SPADES       TEN   OF DIAMONDS
NATURAL
THE PLAYER'S TOTAL IS 1
THE BANKER'S TOTAL IS 9
YOU WON!!!

TOTALS ARE:
COMPUTER HAS 13960
YOU HAVE 6040

WANT TO TRY AGAIN? N
```

# The Program.



A call to RANDOMIZE begins the main program, anticipating the two uses of the RANDOM function in this program—in procedure SHUFFLE, and in the main program itself (when the computer makes a bet). Then, after COMPMONEY and HUMMONEY are both initialized to ten thousand, MAKEDECK is called.

Procedure MAKEDECK initializes the variable DECK. As in other card games in this book, DECK is an array of CARD, and CARD is a record type with two fields:

```
CARD = RECORD
   VALUE: 1..13;
   SUIT: SUITTYPE
END;
```

In Baccarat, however, DECK has 416 elements, to represent eight decks of cards. The VALUE and SUIT fields of each card are assigned values in a **FOR** loop that increments the index (DECKNUM) of DECK from 1 to 416. The VALUE is computed from the formula DECKNUM **MOD** 13 + 1, producing the correct value from 1 to 13 for each card. When all 13 cards of a suit have been assigned values, the suit itself must be changed. This is done in the following lines:

```
IF SUIT <> SPADES THEN
   SUIT := SUCC(SUIT)
ELSE
   SUIT:= HEARTS;
```

Notice that SUIT is declared to be of type SUITTYPE. SUITTYPE is a scalar type:

```
SUITTYPE = (HEARTS,CLUBS,DIAMONDS,SPADES);
```

Thus SUCC(SUIT) produces the *succeeding* value in the range of SUITTYPE.

Back in the main program, CARDNUM (the latest dealt card) is initialized to DECKSIZE. As we will see, this initialization triggers a call to SHUFFLE at the beginning of procedure PLAY.

Next the user inputs a P or a B, and the input is validated:

```
WHILE NOT (CH IN ['P','B']) DO
```

Now the player places a bet against the bank. If the computer is the player, the bet is chosen randomly, but a multiple of 10 is assured:

```
BET := RANDOM MOD START + 10;
BET := BET DIV 10;
BET := BET * 10;
```

(The computer apparently hates to bother with anything less than $10.)

Next, procedure PLAY is called. PLAY begins with some initializations, and then calls SHUFFLE whenever CARDNUM gets toward the end of the deck:

```
IF CARDNUM > (DECKSIZE - 7) THEN
  SHUFFLE;
```

Procedure SHUFFLE goes through the entire deck, exchanging the positions of random pairs of cards. (We have noted elsewhere that SHUFFLE resembles a sort routine, in that it uses a temporary storage variable TMP to implement the exchange.)

Four calls to CARDEAL then deal and display the first four cards of the play. (Notice that the parameter of CARDEAL alternates between PLAYHAND and BANKHAND.)

Procedure CARDEAL stores the present value of the hand in the *variable* parameter ADDTO. (CARDEAL actually calculates the new value of the hand.) The first statement in CARDEAL is a call to SHOWCARD:

```
SHOWCARD(DECK[CARDNUM]);
```

Procedure SHOWCARD thus displays the top card of the deck. It uses two **CASE** statements, one for the value of the card:

```
CASE DISPLAY.VALUE OF . . .
```

and one for the suit:

```
CASE DISPLAY.SUIT OF . . .
```

CARDEAL continues by calculating the real point value of the hand. (The value of the new card is added to the former value of the hand, and factors of ten are eliminated using a **MOD** operation.) Finally, CARDNUM is incremented.

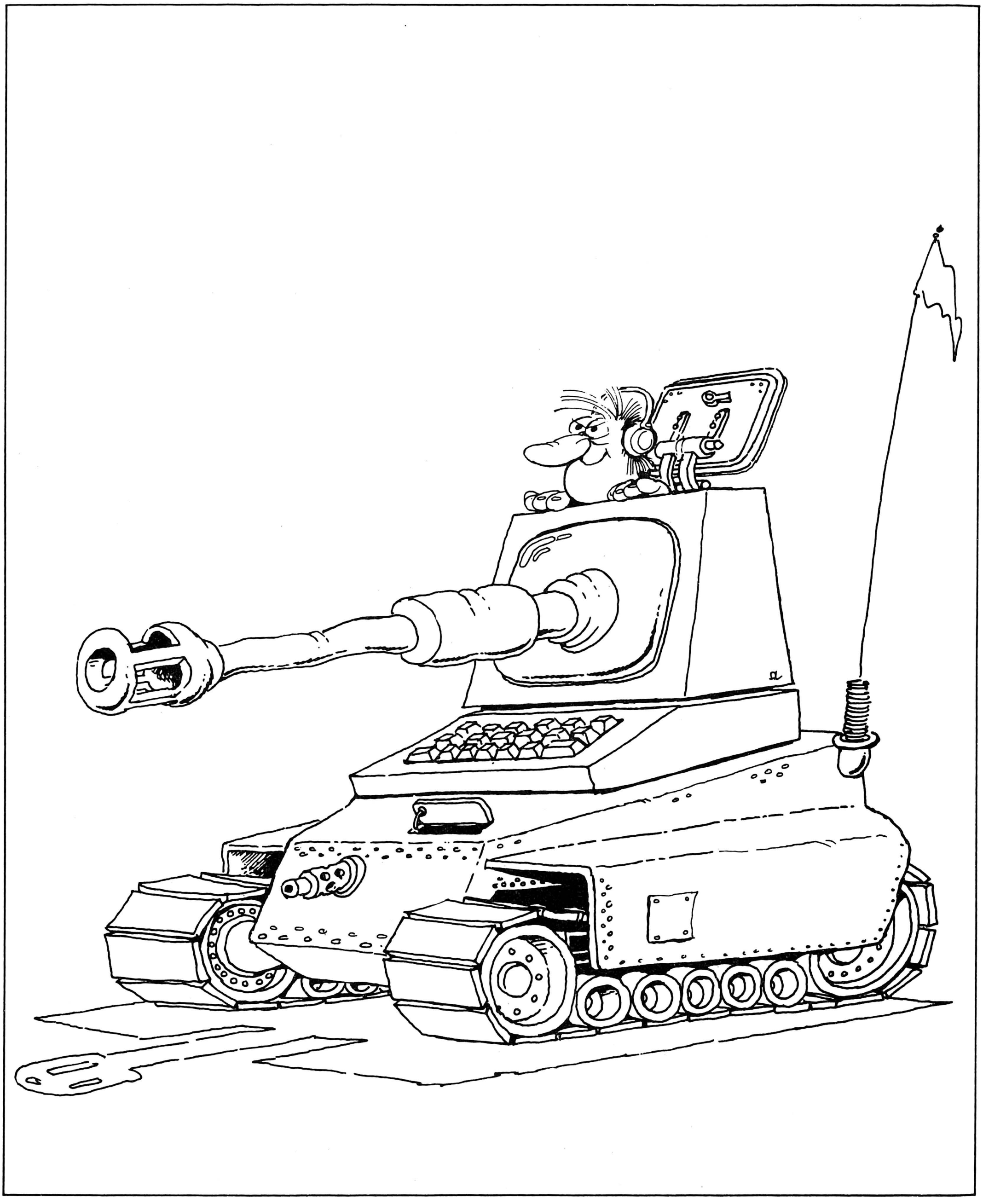
The rest of procedure PLAY is merely a series of **IF** statements that decide who, if anyone, must take a third card. If the player takes a card then function BANKDEAL is called to determine whether or not the banker must also take a third card. The algorithm of BANKDEAL is contained in a **CASE** statement that is dependent on the *value* (LASTVAL) of the third card that was dealt to the player.

Finally, PLAY displays the totals for each hand. The main program then assigns the appropriate hand values to the computer and the human (depending on who is the banker and who is the player), announces the winner, and redistributes the wealth (COMPMONEY and HUMMONEY) accordingly.

```
(*
 * BACCARAT
 * PLAYS THE CARD GAME OF BACCARAT
 *)
PROGRAM BACCARAT (INPUT,OUTPUT);
USES APPLESTUFF;

CONST
   DECKSIZE      =  416;        (* SIZE OF 8 DECKS *)
   START         =10000;        (* THE AMOUNT OF MONEY YOU START WITH *)


TYPE
   SUITTYPE   = (HEARTS,CLUBS,DIAMONDS,SPADES);

   CARD = RECORD
      VALUE : 1..13;            (* CARD NUM *)
      SUIT  : SUITTYPE
   END;

   HANDVALUE = 0..9;            (* POSSIBLE POINTS IN A HAND *)

   DECKTYPE  = 1..DECKSIZE;     (* FOR INDEXES INTO THE DECK *)


VAR
   CARDNUM  : DECKTYPE;         (* CURRENT CARD WE ARE PLAYING *)
   DECK     : ARRAY [DECKTYPE] OF CARD; (* THE DECK *)

   BANKHAND : HANDVALUE;        (* THE BANKER'S HAND *)
   PLAYHAND : HANDVALUE;        (* THE PLAYER'S HAND *)
   COMPHAND : HANDVALUE;        (* VALUE OF THE COMPUTER'S HAND *)
   HUMHAND  : HANDVALUE;        (* THE VALUE OF THE HUMAN'S HAND *)

   COMPMONEY: INTEGER;          (* THE COMPUTER'S MONEY *)
   HUMMONEY : INTEGER;          (* THE PLAYER'S MONEY *)
   BET      : INTEGER;          (* THE SIZE OF THE BET *)

   CH       : CHAR;             (* USED TO ANSWER QUESTIONS *)

   (*
    * MAKEDECK
    * CREATE THE DECK
    *)
```

```
PROCEDURE MAKEDECK;
VAR
   DECKNUM : DECKTYPE;
   SUIT    : SUITTYPE;
   VALUE   : 1..13;
BEGIN
   SUIT := HEARTS;
   FOR DECKNUM := 1 TO DECKSIZE DO
   BEGIN
      DECK[DECKNUM].VALUE := DECKNUM MOD 13 +1;
      DECK[DECKNUM].SUIT := SUIT;
      IF DECK[DECKNUM].VALUE = 13 THEN
         IF SUIT <> SPADES THEN
            SUIT := SUCC(SUIT)
         ELSE
            SUIT := HEARTS
   END
END; (* MAKEDECK *)

(*
 * SHUFFLE
 * GO THROUGH THE DECK EXCHANGING CARDS
 *)
PROCEDURE SHUFFLE;
VAR
   TMP : CARD;
   DECKNUM,
   NEWNUM : DECKTYPE;
BEGIN
   CARDNUM := 1;
   FOR DECKNUM := 1 TO DECKSIZE DO
   BEGIN
      NEWNUM := RANDOM MOD DECKSIZE +1;
      TMP := DECK[NEWNUM];
      DECK[NEWNUM] := DECK[DECKNUM];
      DECK[DECKNUM] := TMP
   END
END; (* SHUFFLE *)

(*
 * SHOWCARD
 * PRINTS OUT A CARD IN A READABLE FORMAT
 *)
PROCEDURE SHOWCARD(DISPLAY:CARD);
BEGIN
   CASE DISPLAY.VALUE OF
      1: WRITE('ACE  ');
      2,3,4,5,6,7,8,9:
         WRITE(DISPLAY.VALUE:1,'    ');
      10:WRITE('TEN  ');
      11:WRITE('JACK ');
      12:WRITE('QUEEN');
      13:WRITE('KING ')
   END;
   WRITE(' OF ');
   CASE DISPLAY.SUIT OF
      HEARTS : WRITE('HEARTS  ');
      DIAMONDS:WRITE('DIAMONDS');
      CLUBS   : WRITE('CLUBS   ');
      SPADES : WRITE('SPADES  ')
   END;
   WRITE('     ')
END; (* SHOWCARD *)

(*
 * PLAY
 * DEAL THE CARDS, THEN DECIDE
 * IF WE NEED TO TAKE ANY CARDS
 *)
PROCEDURE PLAY;
VAR
   STOOD : BOOLEAN;  (* IF THE PLAYER STOOD *)
   (*
    * CARDEAL
    * ACTUALLY DEALS A CARD OUT
    *)
   PROCEDURE CARDEAL(VAR ADDTO:HANDVALUE);
   BEGIN
      SHOWCARD(DECK[CARDNUM]);
      IF DECK[CARDNUM].VALUE <= 9 THEN
         ADDTO := (ADDTO + DECK[CARDNUM].VALUE) MOD 10;
      CARDNUM := CARDNUM + 1

   END;(* CARDEAL *)

(*
 * BANKDEAL
 * DECIDE IF THE BANK SHOULD HAVE A CARD
 * DEALT TO IT
 *)
FUNCTION BANKDEAL:BOOLEAN;
VAR
   LASTVAL : 0..9;        (* THE VALUE OF THE LAST CARD *)
BEGIN
   IF DECK[CARDNUM-1].VALUE >= 10 THEN
      LASTVAL := 0
   ELSE
      LASTVAL := DECK[CARDNUM-1].VALUE;

   BANKDEAL := TRUE;

   CASE LASTVAL OF
      0,9: IF BANKHAND >= 4 THEN
               BANKDEAL := FALSE;

      8:   IF BANKHAND >= 3 THEN
              BANKDEAL := FALSE;

      7,6: IF BANKHAND = 7 THEN
              BANKDEAL := FALSE;
         5,4: IF BANKHAND >= 6 THEN
                 BANKDEAL := FALSE;

         3,2: IF BANKHAND >=5 THEN
                 BANKDEAL := FALSE;

         1:   IF BANKHAND >= 4 THEN
                 BANKDEAL := FALSE
      END
   END; (* BANKDEAL *)


BEGIN
   STOOD := TRUE;
   BANKHAND := 0;
   PLAYHAND := 0;
   IF CARDNUM > (DECKSIZE -7) THEN
      SHUFFLE;

   WRITELN('PLAYER''S HAND        BANKER''S HAND');
   CARDEAL(PLAYHAND);
   CARDEAL(BANKHAND);
   WRITELN;
   CARDEAL(PLAYHAND);
   CARDEAL(BANKHAND);
   WRITELN;

   IF (BANKHAND >=8 ) OR (PLAYHAND >= 8) THEN
      WRITELN('NATURAL')
   ELSE
   BEGIN
      IF PLAYHAND <= 5 THEN
      BEGIN
         STOOD := FALSE;
         WRITELN('THE PLAYER MUST TAKE A CARD');
         CARDEAL(PLAYHAND)
      END;
      WRITELN:
      IF STOOD THEN
      BEGIN
         IF BANKHAND < 6 THEN
         BEGIN
            WRITELN('THE BANKER MUST TAKE A CARD');
            CARDEAL(BANKHAND)
         END
      END
      ELSE
         IF BANKDEAL THEN
         BEGIN
            WRITELN('THE BANKER MUST TAKE A CARD');
            CARDEAL(BANKHAND)
         END;
      WRITELN
   END;
   WRITELN('THE PLAYER''S TOTAL IS ',PLAYHAND);
   WRITELN('THE BANKER''S TOTAL IS ',BANKHAND)
END; (* PLAY *)
```

```
BEGIN
   RANDOMIZE;
   COMPMONEY := START;
   HUMMONEY  := START;
   MAKEDECK;
   CARDNUM := DECKSIZE;
   WHILE CH <> 'N' DO
   BEGIN
      WHILE NOT (CH IN ['P','B']) DO
      BEGIN
         WRITE('DO YOU WANT TO BE THE BANKER, OR PLAYER? ');
         READLN(CH)
      END;
      WRITELN;
      IF CH = 'P' THEN
      BEGIN
         WRITE('BET? ');
         READLN(BET)
      END
      ELSE
      BEGIN
         (*
          * THE COMPUTER IS THE PLAYER
          * SO IT MAKES A RANDOM BET
          *)
         BET := RANDOM MOD START +10;
         BET := BET DIV 10;            (* MAKE SURE THE LAST NUMBER IS 10 *)
         BET := BET * 10;
         WRITELN('THE BET IS ',BET)
      END;
      PLAY;
      RANDOMIZE;
      IF CH = 'B' THEN
      BEGIN
         COMPHAND := PLAYHAND;
         HUMHAND  := BANKHAND
      END
      ELSE
      BEGIN
         COMPHAND := BANKHAND;
         HUMHAND  := PLAYHAND
      END;
      IF BANKHAND = PLAYHAND THEN
         WRITELN('TIE')
      ELSE

      BEGIN
         IF COMPHAND > HUMHAND THEN
         BEGIN
            WRITELN('THE COMPUTER HAS WON');
            COMPMONEY := COMPMONEY + BET;
            HUMMONEY  := HUMMONEY  - BET
         END
         ELSE
         BEGIN
            WRITELN('YOU WON!!!');
            COMPMONEY := COMPMONEY - BET;
            HUMMONEY := HUMMONEY + BET
         END
      END;
      WRITELN;
      WRITELN('TOTALS ARE:');
      WRITELN('COMPUTER HAS ',COMPMONEY);
      WRITELN('YOU HAVE ',HUMMONEY);
      WRITELN;
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
   END;

END.
```



# DODGE

**The Game.** In this game the computer puts you inside a "battlefield" that is 30 spaces long and 15 lines wide. In there with you is a mixed crowd of attackers, mines, and a tank, and your goal is to destroy all the attackers before they destroy you.

Here is a rundown of the characters in this game:

- One tank, represented by a T.
- Six attackers (at the beginning of the game), represented by dollar signs ($).
- Fifteen mines *inside* the field, represented by Xs. In addition, the entire perimeter of the field is made up of mines.
- You are represented by an asterisk (*).

At each turn you will be asked:

```
DIRECTION?
```

and you respond with a number from 0 to 9 to indicate your move. Each time you move, you must try to avoid hitting—or being hit by—an attacker, a mine, or a tank, for any one of them will destroy you. If you want to stay right where you are, type a 0. On the other hand, if you find yourself in a tight spot and you have no safe direction to move, you may type a 9. This command will move you at random to a new location on the field. (You may only have two 9s during the game. Be aware that a 9 command is a calculated risk, because you *might* land on an attacker, a mine, or on the tank.)

The commands 1 through 8 will move you one space in the direction indicated by this key:

8 — NW 1 — N 2 — NE
7 — W 3 — E
6 — SW 5 — S 4 — SE

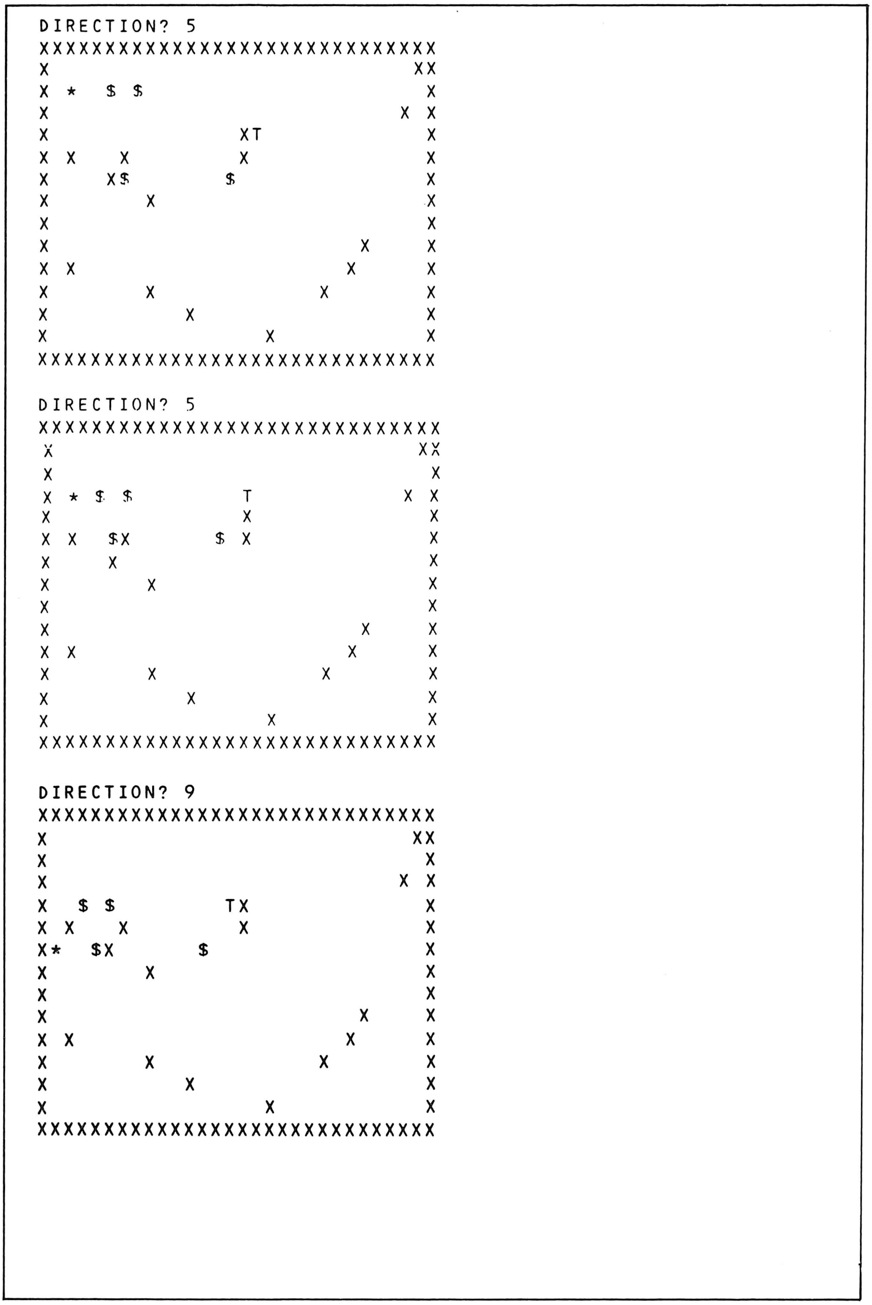
Each time you move, all the attackers and the tank move one space toward you. They will always move to the space that brings them closest to you, no matter what is in their path. Thus, with skill and a bit of luck, you can manipulate the fate of the attackers by luring them toward you and intentionally leading them into a path of destruction. An attacker is eliminated if it collides with a mine or with the tank. If two attackers hit each other, they are both moved randomly to a new spot in the field (after which they both take an extra move toward you). The tank destroys anything in its path—mines, attackers, or you.

The battlefield will be displayed for you after each move, and you will be informed of the demise of any of the characters. The game continues until you have eliminated all six of the attackers—or until they have eliminated you!

```
WANT INSTRUCTIONS? Y
 DODGE
 YOU ARE IN A 30 X 15 SQUARE. THERE ARE
 6 ATTACKERS, ONE TANK, AND YOU. IF YOU
 HIT ONE OF THE ATTACKERS (OR THE TANK ), YOU ARE
 DESTROYED. IF TWO ATTACKERS HIT EACH
 OTHER, THEY ARE RANDOMLY PUT SOMEWHERE
 ELSE. IF AN ATTACKER HITS THE TANK,
 THEN THE ATTACKER IS DESTROYED.
 AS AN ADDED BONUS, THERE ARE 15 MINES
 IN THE SQUARE (AND THE BORDER OF THE
 SQUARE). IF YOU HIT A MINE, YOU ARE
 DESTROYED. IF AN ATTACKER HITS A MINE,
 THEN THE ATTACKER IS DESTROYED. YOUR
 OBJECT IS TO DESTROY THE ATTACKERS.
 YOU CAN MOVE BY
   812
   703
   654
 NOTE THAT 0 LEAVES YOU IN THE SAME PLACE.
 A MOVE OF 9 WILL PUT YOU IN SOME RANDOM
 PLACE ON THE SQUARE (EVEN ON A MINE)
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                   $        X
X                            X
X                 X          X
X     X X                    X
X          X                 X
X                X  X        X
X                 X          X
X X           X    $      X  X
X                            X
X           $         *T     X
X             X    $    $    X
X              X    $   X    X
X                   X  X     X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
DIRECTION? 9

WELL, YOU GOT YOURSELF KILLED!!

WANT TO TRY AGAIN? Y
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X *                        X X
X              X             X
X X   X        X             X
X    X                       X
X       X     $              X
X           $                X
X                       X    X
X X    $               X     X
X     $ X            X T     X
X          X                 X
X            $   X   $       X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
DIRECTION? 0
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X *                        X X
X              X             X
X X   X        X             X
X    X       $               X
X       X  $                 X
X                            X
X     $                 X    X
X X  $                TX     X
X       X            X       X
X          X$       $        X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 8
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X*                           X
X                          X X
X              X             X
X X   X     $  X             X
X    X    $                  X
X       X                    X
X    $                       X
X   $                T  X    X
X X                    X     X
X       X  $       $ X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 2
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X *                         XX
X                            X
X                          X X
X          $   X             X
X X   X  $     X             X
X    X                       X
X   $   X                    X
X  $                T        X
X                       X    X
X X       $       $    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
DIRECTION? 3
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X  *                        XX
X                            X
X         $                X X
X       $      X             X
X X   X        X             X
X  $ X                       X
X  $    X          T         X
X                            X
X        $       $      X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 7
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X *                         XX
X        $                   X
X      $                   X X
X              X             X
X X   X        X             X
X $  X            T          X
X       X                    X
X       $       $            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 0
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X *     $                   XX
X     $                      X
X                          X X
X              X             X
X X   X        X T           X
X    X                       X
X      $X      $             X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

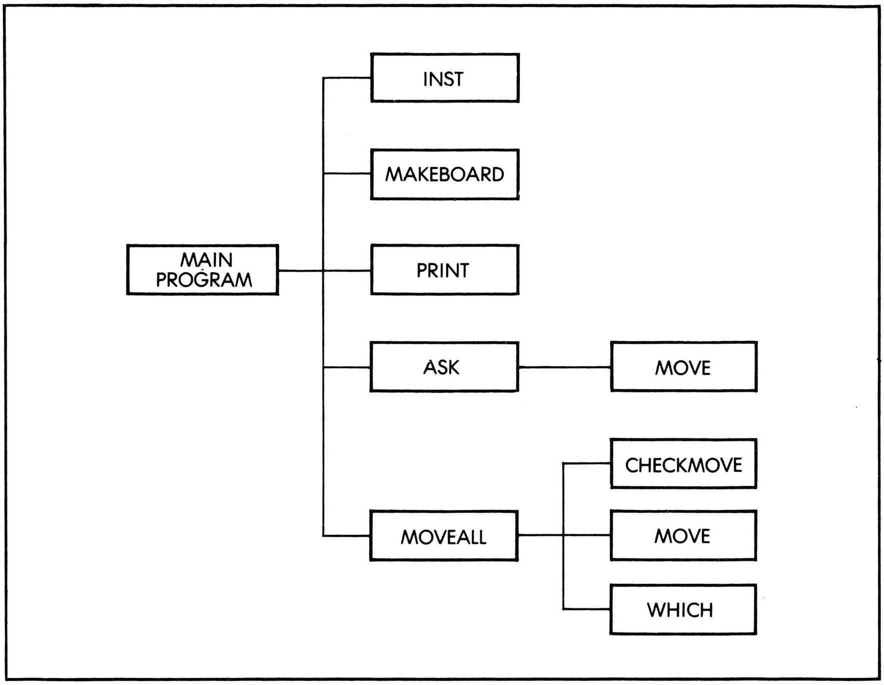


```
DIRECTION? 5
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X *  $ $                     X
X                          X X
X              XT            X
X X   X        X             X
X    X$       $              X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 5
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X * $ $        T           X X
X              X             X
X X  $X      $ X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 9
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X  $ $        TX             X
X X   X        X             X
X*  $X      $                X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
DIRECTION? 0
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X X $ X      T X             X
X* $ X     $                 X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 1
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X*X$  X   $ T  X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 0
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X*X   X  $ T   X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
DIRECTION? 0
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X*X   X $ T    X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 0
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X*X   X$ T     X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DIRECTION? 0
AN ATTACKER JUST HIT A MINE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                           XX
X                            X
X                          X X
X              X             X
X*X   X T      X             X
X    X                       X
X       X                    X
X                            X
X                       X    X
X X                    X     X
X       X            X       X
X          X                 X
X                X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

CONGRATULATIONS, YOU DID IT!!!

WANT TO TRY AGAIN? N
```

# The Program.


INST
MAKEBOARD
MAIN PROGRAM
PRINT
ASK
MOVE
CHECKMOVE
MOVEALL
MOVE
WHICH

Three important type declarations appear in the program heading of DODGE. WHAT and DIRECTION are scalar types. WHAT defines the possible values of each position in the battlefield:

```
WHAT = (ME, ATTACKER, TANK, MINE, NOTHING);
```

(The variable SQUARE, a two-dimensional array of WHAT, represents the field.) DIRECTION represents the range of move possibilities:

```
DIRECTION = (STAY, N, NE, E, SE, S, SW, W, NW,
                NOWHERE);
```

Finally, LOC is a record type with two fields to represent the x and *y* coordinates of each position in the square:

```
LOC = RECORD
   XCORD: XINDEX;
   YCORD: YINDEX;
END;
```

TANKLOC and MYLOC, representing the positions of the tank and the player, are variables of type LOC. ATACK, representing the positions of the attackers, is an array of LOC.

The main program calls INST to provide instructions and then enters a **REPEAT/UNTIL** loop that keeps starting new games as long as the player wants to continue. A call to RANDOMIZE anticipates use of the RANDOM function. Then MAKEBOARD is called to set the initial positions in the square, and PRINT is called to display the square.

Procedure MAKEBOARD begins with several initializations. Variables NINES (the number of 9 commands the player has) and LEFT (the number of attackers) are both set to their maximum values (2 and 6 respectively); the value of these variables will be decremented during the course of the game. All the elements of SQUARE are initialized to NOTHING by a pair of nested **FOR** loops. The next two **FOR** loops set the perimeter of the square; all the values of the perimeter elements are set to MINE.

Then, in four separate **WHILE** loops, the mines, attackers, tank, and player (ME) are assigned their initial locations in SQUARE. In each case, the (x,y) coordinates are determined by using the RANDOM function:

```
XLOC := 1 + RANDOM MOD LENGTH;
YLOC := 1 + RANDOM MOD WIDTH;
```

The position is examined to make sure it does not already contain a character:

```
IF SQUARE[YLOC][XLOC] = NOTHING
```

and finally, the assignment is made:

```
SQUARE[YLOC][XLOC] := ATTACKER;
```

The record variables TANKLOC and MYLOC, and the array of records ATACK are all assigned values in these loops also.

Procedure PRINT contains a **CASE** statement inside two levels of **FOR** loops, to display each element of SQUARE, character by character:

```
CASE SQUARE[J][I] OF
   ME: WRITE('*');
   TANK: WRITE('T');
   ATTACKER: WRITE('$');
   MINE: WRITE('X');
   NOTHING: WRITE (' ');
END;
```

Next in the main program, a **WHILE** loop initiates the actual play of the game, and continues the action until the player is destroyed (represented by setting MYLOC.XCORD to 0) or no attackers are left:

```
WHILE (MYLOC.XCORD <> 0) AND (LEFT > 0) DO
```

Function ASK (which returns a value of type WHAT) is called to get the player's move; if any value other than NOTHING is returned (i.e., if the player has hit something) then it is curtains for the player:

```
IF ASK <> NOTHING THEN
   MYLOC.XCORD = 0
```

Before function ASK prompts for the input, it initializes COMMAND (an array of DIRECTION) so that the indices (the *characters* '0' to '9') will correspond to the appropriate values of DIRECTION. The **FOR** loop that performs this task makes use of the SUCC function to "increment" the values of DIR:

```
DIR := SUCC(DIR)
```

Finally the player is asked for a command and the input is validated. Notice that if the 9 command is requested, the validation statement checks to see if the player still has NINES options left:

```
IF NOT(CH IN ['0'..'9']) OR ((CH='9') AND (NINES=0))
```

If the input is not appropriate, an error message is displayed and ASK is called recursively:

```
ASK := ASK;
```

ASK contains the algorithm for handling the 9 command. Since the player is moving to a new position, the old position of SQUARE is set to NOTHING. New values for MYLOC are determined via calls to RANDOM, and ME is assigned to the elements of SQUARE determined by the new coordinates. (CH is set to '0' to prepare for the call to MOVE.)

Whatever the command, ASK calls function MOVE to determine the value of the position to which the player is moving. The parameters of the call are the coordinates of the player's old position, and the command character:

```
ASK := MOVE(MYLOC.XCORD, MYLOC.YCORD,
            COMMAND[CH]);
```

Function MOVE uses a **CASE** statement to increment or decrement the coordinates appropriately for the player's move, and returns the value of the element of SQUARE represented by the new coordinates.

The game continues with calls (from the main program) to MOVEALL and PRINT.

Procedure MOVEALL contains instructions for moving the attackers and the tank toward the player. The remaining attackers are moved one by one in a **FOR** loop. First the old position (in SQUARE) of each attacker is set to NOTHING. Then, in an efficient sequence of nested procedure calls, the best move is determined (WHICH), the move is made (MOVE) and the results are investigated (CHECKMOVE):

```
CHECKMOVE(I, MOVE(ATACK[I].XCORD,ATACK[I].YCORD,
          WHICH(ATACK[I])));
```

Through a series of comparisons between the values of MYLOC and the current location of the attacker (CURR), function WHICH finds the single move that will carry the attacker closest to the player. The mid-directions (NW, SW, NE, SE) are determined in a **CASE** statement. We have already seen how function MOVE actually implements the move and returns the present value (of type WHAT) of the position that is being approached.

Procedure CHECKMOVE, which is used only for moving attackers, receives the index (I) into ATACK that indicates which attacker is moving, and the present value of the position that attacker is moving to (DIR, of type WHAT). CHECKMOVE uses a **CASE** statement (DIR is the case "selector") to decide what to do. If the position has a value of NOTHING, the attacker is moved there. If, on the other hand, a TANK or a MINE occupies the position, the attacker is eliminated:

```
ATACK[I].XCORD := 0;
```

A postmortem message is displayed and the number of remaining attackers is decremented:

```
LEFT := LEFT - 1;
```

If the attacker has hit the player (ME), then the player is wiped out, and a fairly undignified obituary is written:

```
WRITELN('YOU JUST DIED!!! ');
MYLOC.XCORD := 0;
```

Finally, the most complicated case is when an attacker hits an attacker. First the attacker that has been hit must be identified. This is done by comparing the (x,*y*) coordinates of the moving attacker with the present positions of all the other attackers until a match is found. That position is set to NOTHING, and the two attackers are given new positions (via calls to the RANDOM function). Then, to make the game even more interesting, each of the two attackers is moved *again*, one position toward the player. The nested procedure calls that we have already seen in MOVEALL are used once again, but this time the call to CHECKMOVE is recursive.

MOVEALL must also move the tank. A **CASE** statement is used to destroy whatever is in the tank's way. A call to MOVE provides the case selector, and a nested call to WHICH determines the best move:

```
CASE MOVE(TANKLOC.XCORD, TANKLOC.YCORD,
          WHICH(TANKLOC)) OF
```

Once MOVEALL has determined all the action, PRINT displays the new battlefield, and the game continues.

```
(*
 * DODGE
 * YOU ARE IN A 30 X 15 SQUARE. THERE ARE
 * 6 ATTACKERS, ONE TANK, AND YOU. IF YOU
 * HIT ONE OF THE ATTACKERS (OR THE TANK ), YOU ARE
 * DESTROYED. IF TWO ATTACKERS HIT EACH
 * OTHER, THEY ARE RANDOMLY PUT SOMEWHERE
 * ELSE. IF AN ATTACKER HITS THE TANK,
 * THEN THE ATTACKER IS DESTROYED.
 * AS AN ADDED BONUS, THERE ARE 15 MINES
 * IN THE SQUARE (AND THE BORDER OF THE
 * SQUARE). IF YOU HIT A MINE, YOU ARE
 * DESTROYED. IF AN ATTACKER HITS A MINE,
 * THEN THE ATTACKER IS DESTROYED. YOUR
 * OBJECT IS TO DESTROY THE ATTACKERS.
 * YOU CAN MOVE BY
 *   812
 *   703
 *   654
 * NOTE THAT 0 LEAVES YOU IN THE SAME PLACE.
 * A MOVE OF 9 WILL PUT YOU IN SOME RANDOM
 * PLACE ON THE SQUARE (EVEN ON A MINE)
 *)
PROGRAM DODGE (INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   NUMATTACKERS    =  6;          (* NUMBER OF ATTACKERS *)
   LENGTH          = 30;          (* LENGTH OF PLAYING FIELD *)
   WIDTH           = 15;          (* WIDTH  "      "       "   *)
   NUMINES         = 15;          (* NUMBER OF MINES *)
   NINEMAX         =  2;          (* NUMBER OF TIMES THE 9 ESCAPE CAN BE USED *)

TYPE
   (* WHAT CAN BE AT A LOCATION *)
   WHAT = (ME,ATTACKER,TANK,MINE,NOTHING);

   XINDEX = 0..LENGTH;            (* THE TYPE OF X INDICES *)
   YINDEX = 0..WIDTH;             (* THE TYPE OF Y INDICES *)

   (*
    * DIRECTIONS SOMETHING CAN MOVE
    *)
   DIRECTION = (STAY,N,NE,E,SE,S,SW,W,NW,NOWHERE);

   (*
    * THE LOCATION OF AN OBJECT
    *)
   LOC   = RECORD
      XCORD : XINDEX;             (* 0 INDICATES DEAD *)
      YCORD : YINDEX
   END;
```

```
VAR
   (* THE PLAYING FIELD *)
   SQUARE : ARRAY[1..WIDTH,1..LENGTH]OF WHAT;

   TANKLOC: LOC;                  (* LOCATION OF THE TANK *)
   MYLOC  : LOC;                  (* MY LOCATION *)
   ATACK  : ARRAY[1..NUMATTACKERS]OF LOC; (* THE ATTACKERS *)

   CH     : CHAR;                 (* USED TO ANSWER QUESTIONS *)
   LEFT   : 0..NUMATTACKERS;      (* NUMBER OF ATTACKERS LEFT *)
   NINES  : 0..NINEMAX;           (* NUMBER OF 9'S YOU HAVE DONE *)

   (*
    * MAKEBOARD
    * CREATE THE BOARD, AND PLACE EVERY THING
    * IN AT RANDOM PLACES
    *)
   PROCEDURE MAKEBOARD;
   VAR
      PLACED : INTEGER;           (* NUMBER PLACED *)
      XLOC   : XINDEX;            (* X LOCATION OF AN OBJECT *)
      YLOC   : YINDEX;            (* Y LOCATION *)
      I,J    : INTEGER;           (* INDEX VARIABLES *)

   BEGIN
      NINES := NINEMAX;
      FOR I := 1 TO WIDTH DO
         FOR J := 1 TO LENGTH DO
            SQUARE[I][J] := NOTHING;
      LEFT := NUMATTACKERS;

      FOR I := 1 TO LENGTH DO
      BEGIN
         SQUARE[1][I] := MINE;
         SQUARE[WIDTH][I] := MINE
      END;

      FOR I := 1 TO WIDTH DO
      BEGIN
         SQUARE[I][1] := MINE;
         SQUARE[I][LENGTH] := MINE
      (*
       * PLACE THE MINES
       *)
      PLACED := 0;
      WHILE PLACED < NUMINES DO
      BEGIN
         XLOC := 1 + RANDOM MOD LENGTH;
         YLOC := 1 + RANDOM MOD WIDTH;
         IF SQUARE[YLOC][XLOC] = NOTHING THEN
         BEGIN
            SQUARE[YLOC][XLOC] := MINE;
            PLACED := PLACED +1
         END
      END;
    PLACED := 1;
    (*
     * PLACE THE ATTACKERS
     *)
    WHILE PLACED <= NUMATTACKERS DO
    BEGIN
       XLOC := 1 + RANDOM MOD LENGTH;
       YLOC := 1 + RANDOM MOD WIDTH;
       IF SQUARE[YLOC][XLOC] = NOTHING THEN
       BEGIN
          SQUARE[YLOC][XLOC] := ATTACKER;
          ATACK[PLACED].XCORD := XLOC;
          ATACK[PLACED].YCORD := YLOC;
          PLACED := PLACED +1
       END
    END;

    WHILE SQUARE[YLOC][XLOC] <> NOTHING DO
    BEGIN
       XLOC := 1 + RANDOM MOD LENGTH;
       YLOC := 1 + RANDOM MOD WIDTH
    END;
    SQUARE[YLOC][XLOC] := TANK;
    TANKLOC.XCORD := XLOC;
    TANKLOC.YCORD := YLOC;


    WHILE SQUARE[YLOC][XLOC] <> NOTHING DO
    BEGIN
       XLOC := 1 + RANDOM MOD LENGTH;
       YLOC := 1 + RANDOM MOD WIDTH
    END;
    SQUARE[YLOC][XLOC] := ME;
    MYLOC.XCORD := XLOC;
    MYLOC.YCORD := YLOC;

END; (* MAKEBOARD *)

(*
 * PRINT
 * PRINT OUT THE FIELD
 *)
PROCEDURE PRINT;
VAR
   I : XINDEX;                (* INDEX FOR LENGTH *)
   J : YINDEX;                (* INDEX FOR WIDTH *)
BEGIN
   FOR J := 1 TO WIDTH DO
   BEGIN
      FOR I := 1 TO LENGTH DO
         CASE SQUARE[J][I] OF

            ME : WRITE('*');

            TANK: WRITE('T');

            ATTACKER: WRITE('$');
```

```
              MINE:WRITE('X');

              NOTHING:WRITE(' ')

           END;
        WRITELN
     END
  END; (* PRINT *)


  (*
   * MOVE
   * ATTEMPT TO MOVE SOMETHING
   * FROM CURX,CURY, IN THE DIRECTION
   * INDICATED. RETURNS WHAT WAS ON THE
   * LOCATION
   *)
  FUNCTION MOVE(VAR CURX:XINDEX;VAR CURY:YINDEX ; WHERE : DIRECTION):WHAT;
  BEGIN
     CASE WHERE OF

        N    : CURY := CURY -1;

        STAY: ;

        S    : CURY := CURY +1;

        W    : CURX := CURX -1;

        E    : CURX := CURX +1;

        NW   : BEGIN
                  CURX := CURX -1;
                  CURY := CURY -1
               END;
        SW   : BEGIN
                  CURX := CURX -1;
                  CURY := CURY +1
               END;
        NE   : BEGIN
                  CURX := CURX +1;
                  CURY := CURY -1
               END;
        SE   : BEGIN
                  CURX := CURX +1;
                  CURY := CURY +1
               END
     END;

     MOVE := SQUARE[CURY][CURX]
  END; (* MOVE *)
```

```
(*
 * ASK
 * ASK THE USER WHERE HE WANTS
 * TO GO, THEN ATTEMPT TO GO THERE.
 * RETURNS WHAT THE PLAYER LANDS ON.
 *)
FUNCTION ASK:WHAT;
VAR
   COMMAND:ARRAY ['0'..'9'] OF DIRECTION; (* COMMAND LIST *)
   DIR    : DIRECTION;
   CH     :CHAR;                      (* CHARACTER FROM USER *)
BEGIN
   (*
    * INIT THE COMMANDS
    *)
   DIR := STAY;
   FOR CH := '0' TO '9' DO
   BEGIN
      COMMAND[CH] := DIR;
      DIR := SUCC(DIR)
   END;

   WRITE('DIRECTION? ');
   READLN(CH);
   IF NOT (CH IN ['0'..'9']) OR ((CH = '9') AND (NINES = 0))THEN
   BEGIN
      WRITELN('HOW IS THAT?');
      ASK := ASK
   END

   ELSE
   BEGIN
      SQUARE[MYLOC.YCORD][MYLOC.XCORD] := NOTHING;

      (*
       * ON A COMMAND OF '9', RELOCATE
       * YOURSELF RANDOMLY
       *)
      IF CH = '9' THEN
      BEGIN
         NINES := NINES - 1;
         MYLOC.XCORD := 1 + RANDOM MOD LENGTH;
         MYLOC.YCORD := 1 + RANDOM MOD WIDTH;
         CH := '0'
      END;
      ASK := MOVE(MYLOC.XCORD,MYLOC.YCORD,COMMAND[CH]);
      SQUARE[MYLOC.YCORD][MYLOC.XCORD] := ME
   END
END; (* ASK *)

(*
 * MOVEALL
 * MOVES ALL THE ATTACKERS AROUND, AND
 * THE TANK
 *)
```

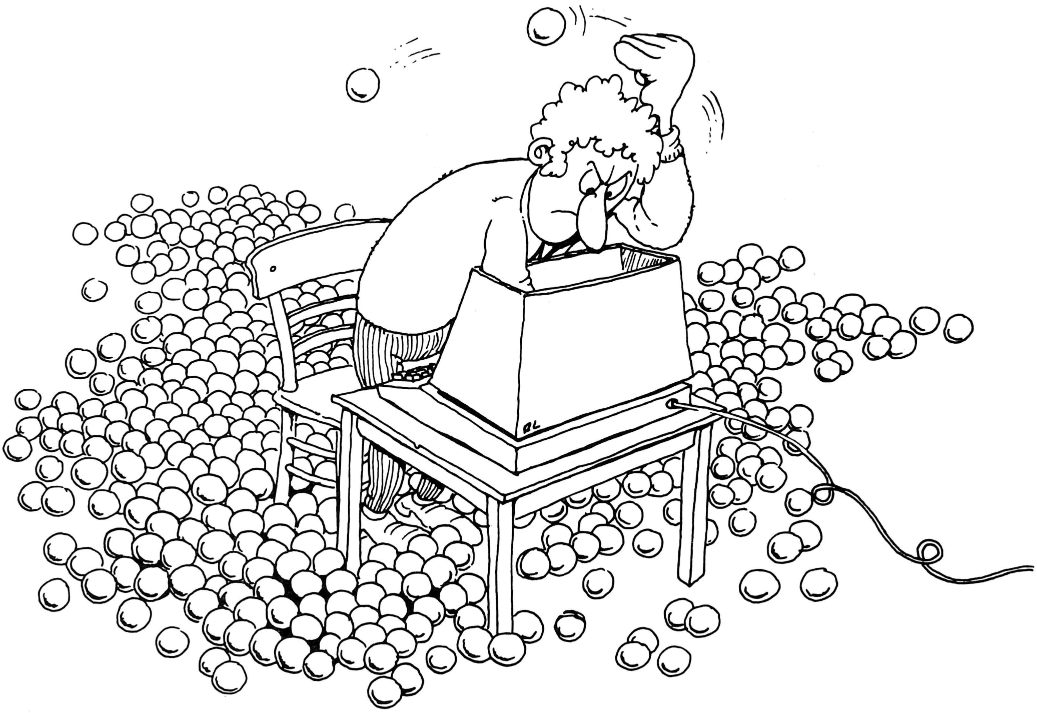
```
PROCEDURE MOVEALL;
VAR
   I    : 1..NUMATTACKERS;   (* INDEX USED TO RUN THROUGH ALL ATTACKERS *)

   (*
    * WHICH
    * GIVEN A LOC, IT DECIDES WHICH DIRECTION
    * TO GO SO AS TO COME CLOSER TO YOU
    * RETURNS THE DIRECTION
    *)
   FUNCTION WHICH(CURR : LOC):DIRECTION;
   VAR
      XDIR,                  (* BEST X DIRECTION *)
      YDIR : DIRECTION;      (* BEST Y DIRECTION *)
   BEGIN
      XDIR := STAY;
      YDIR := STAY;

      IF MYLOC.XCORD > CURR.XCORD THEN
         XDIR := E
      ELSE
         IF MYLOC.XCORD < CURR.XCORD THEN
            XDIR := W;

      IF MYLOC.YCORD > CURR.YCORD THEN
         YDIR := S
      ELSE
         IF MYLOC.YCORD < CURR.YCORD THEN
            YDIR := N;

      IF (XDIR = STAY) OR (YDIR = STAY ) THEN
      BEGIN
         IF XDIR = STAY THEN
            WHICH := YDIR
         ELSE
            WHICH := XDIR
      END
      ELSE
      BEGIN
         CASE XDIR OF
            W: IF YDIR = N THEN
                  WHICH := NW
               ELSE
                  WHICH := SW;
            E: IF YDIR = N THEN
                  WHICH := NE
               ELSE
                  WHICH := SE
         END
      END
   END; (* WHICH *)
```

```
(*
 * CHECKMOVE
 * CHECKS TO SEE WHAT HAPPENS TO
 * AN ATTACKER WHEN IT IS MOVED
 *)
PROCEDURE CHECKMOVE(I : INTEGER; DIR : WHAT);
VAR
   REMEMBER,                     (* BOTH USED WHEN 1 ATTACKER HITS ANOTHER *)
   INDEX   : 1..NUMATTACKERS;
BEGIN
   CASE DIR OF
      NOTHING : SQUARE[ATACK[I].YCORD][ATACK[I].XCORD] := ATTACKER;

      TANK    :  BEGIN
                    WRITELN('THE TANK JUST DESTROYED AN ATTACKER');
                    ATACK[I].XCORD := 0;
                    LEFT := LEFT -1
                 END;

      ME       :  BEGIN
                     WRITELN('YOU JUST DIED!!! ');
                     MYLOC.XCORD := 0
                  END;

      ATTACKER  : BEGIN
                     (*
                      * FIND OUT WHICH ATTACKER WE HIT
                      *)

                     FOR INDEX := 1 TO NUMATTACKERS DO
                        IF (ATACK[I].XCORD = ATACK[INDEX].XCORD) AND
                           (ATACK[I].YCORD = ATACK[INDEX].YCORD) AND
                           (I <> INDEX) THEN
                              REMEMBER := INDEX;
                     WRITELN('TWO ATTACKERS JUST HIT');

                     (* REMOVE IT FROM THE FIELD *)
                     SQUARE[ATACK[REMEMBER].YCORD][ATACK[REMEMBER].XCORD]
                        := NOTHING;

                     (* CHOOSE A RANDOM SPOT FOR THE FIRST ATTACKER *)

                     ATACK[I].XCORD := 1 + RANDOM MOD LENGTH;
                     ATACK[I].YCORD := 1 + RANDOM MOD WIDTH;
                     (* MOVE THE ATTACKER *)
                     CHECKMOVE(I,MOVE(ATACK[I].XCORD,ATACK[I].YCORD,
                               WHICH(ATACK[I])));

                     (* REPEAT THE SAME WITH THE OTHER ATTACKER *)
                     ATACK[REMEMBER].XCORD := 1 + RANDOM MOD LENGTH;
                     ATACK[REMEMBER].YCORD := 1 + RANDOM MOD WIDTH;
                     CHECKMOVE(REMEMBER,MOVE(ATACK[REMEMBER].XCORD,
                     ATACK[REMEMBER].YCORD,
                               WHICH(ATACK[REMEMBER])));

                 END;
```

```
         MINE     :  BEGIN
                        WRITELN('AN ATTACKER JUST HIT A MINE');
                        ATACK[I].XCORD := 0;
                        LEFT := LEFT -1
                     END
      END;
   END; (* CHECKMOVE *)

BEGIN
   (*
    * FOR EACH ATTACKER, IF HE IS STILL
    * ALIVE, MOVE HIM TOWARD YOU
    *)

   FOR I := 1 TO NUMATTACKERS DO
   BEGIN
      IF ATACK[I].XCORD <> 0 THEN
      BEGIN
         SQUARE[ATACK[I].YCORD][ATACK[I].XCORD] := NOTHING;
         CHECKMOVE(I,MOVE(ATACK[I].XCORD,ATACK[I].YCORD,WHICH(ATACK[I])))
      END
   END;

   (*
    * MOVE THE TANK, NOTE THAT THE TANK
    * WILL DESTROY ANYTHING IN ITS WAY
    * (INCLUDING MINES)
    *)
   SQUARE[TANKLOC.YCORD][TANKLOC.XCORD] := NOTHING;

   CASE MOVE(TANKLOC.XCORD,TANKLOC.YCORD,WHICH(TANKLOC)) OF
      ATTACKER : BEGIN
                    WRITELN(' THE TANK JUST DESTROYED AN ATTACKER');
                    LEFT := LEFT -1;
                    FOR I := 1 TO NUMATTACKERS DO
                       IF ATACK[I] = TANKLOC THEN
                          ATACK[I].XCORD := 0
                 END;

      MINE     : WRITELN('THE TANK JUST DESTROYED A MINE');

      ME       : BEGIN
                    WRITELN('THE TANK JUST DESTROYED YOU');
                    MYLOC.XCORD := 0
                 END;

      NOTHING  :
   END;
   SQUARE[TANKLOC.YCORD][TANKLOC.XCORD] := TANK
END; (* MOVEALL *)

(*
 * INST
 * PRINTS OUT THE INSTRUCTIONS
 *)
```

```
  PROCEDURE INST;
  BEGIN
    WRITELN(' DODGE');
    WRITELN(' YOU ARE IN A 30 X 15 SQUARE. THERE ARE');
    WRITELN(' ',NUMATTACKERS:1,' ATTACKERS, ONE TANK, AND YOU. IF YOU');
    WRITELN(' HIT ONE OF THE ATTACKERS (OR THE TANK ), YOU ARE ');
    WRITELN(' DESTROYED. IF TWO ATTACKERS HIT EACH');
    WRITELN(' OTHER, THEY ARE RANDOMLY PUT SOMEWHERE');
    WRITELN(' ELSE. IF AN ATTACKER HITS THE TANK,');
    WRITELN(' THEN THE ATTACKER IS DESTROYED.');
    WRITELN(' AS AN ADDED BONUS, THERE ARE ',NUMINES,' MINES');
    WRITELN(' IN THE SQUARE (AND THE BORDER OF THE ');
    WRITELN(' SQUARE). IF YOU HIT A MINE, YOU ARE');
    WRITELN(' DESTROYED. IF AN ATTACKER HITS A MINE,');
    WRITELN(' THEN THE ATTACKER IS DESTROYED. YOUR');
    WRITELN(' OBJECT IS TO DESTROY THE ATTACKERS.');
    WRITELN(' YOU CAN MOVE BY');
    WRITELN('   812');
    WRITELN('   703');
    WRITELN('   654');
    WRITELN(' NOTE THAT 0 LEAVES YOU IN THE SAME PLACE.');
    WRITELN(' A MOVE OF 9 WILL PUT YOU IN SOME RANDOM');
    WRITELN(' PLACE ON THE SQUARE (EVEN ON A MINE)');
  END; (* INST *)


(* DODGE *)
BEGIN
  WRITE('WANT INSTRUCTIONS? ');
  READLN(CH);
  IF CH = 'Y' THEN
    INST;

  REPEAT
    RANDOMIZE;

    MAKEBOARD;
    PRINT;
    WHILE (MYLOC.XCORD <> 0 ) AND (LEFT > 0) DO
    BEGIN
      (* TEST TO SEE IF YOU KILLED YOURSELF *)
      IF ASK <> NOTHING THEN
        MYLOC.XCORD := 0
      ELSE
      BEGIN
        MOVEALL;
        PRINT
      END;
      WRITELN;
    END;
```

```
      IF MYLOC.XCORD = 0 THEN
         WRITELN('WELL, YOU GOT YOURSELF KILLED!!')
      ELSE
         WRITELN('CONGRATULATIONS, YOU DID IT!!!');

      WRITELN;
      WRITE('WANT TO TRY AGAIN? ');
      READLN(CH)
   UNTIL CH <> 'Y';

END. (* DODGE *)
```



BL

# BLACKBOX

**The Game.** The object of this game is to guess the location of the balls that are placed in a box partitioned 8 by 8. At the beginning of the game you specify how many balls you want to put into the box. The computer places each of them randomly in one of the 64 compartments of the grid.

You can gain information about the location of the balls by shooting "rays" through the box and studying how these rays are deflected or absorbed. In response to the question:

```
WHERE SHOULD I SHOOT NOW?
```

you may identify the starting point of a ray by entering a number from 1 to 32. The first time you see the box you will understand the significance of these numbers: 1 identifies the left side of the first row, and the numbers are incremented going counterclockwise around the box. (Thus, numbers 1 to 8 identify the left side of the rows; 17 to 24 the right side of the rows; 9 to 16 the bottom of the columns; 25 to 32 the top of the columns.)

If a travelling ray does not come near a ball, the ray's path will continue straight along a row or a column, clear to the opposite side of the box. However, several things can happen if a ray approaches a ball:

- If the ray hits a ball straight on, the ray will be absorbed.
- If the ray goes between two balls that are separated by only one compartment of the grid, the ray will be reflected back to its source.
- If the source of the ray is directly adjacent to a ball located on the edge of the box, the ray will be reflected back to its source.
- If the ray enters a compartment (of the grid) that is diagonal to a compartment containing a ball, the ray will be deflected at a right angle to its original path.

Shortly after you shoot a ray through the box, you will see one of two messages; either:

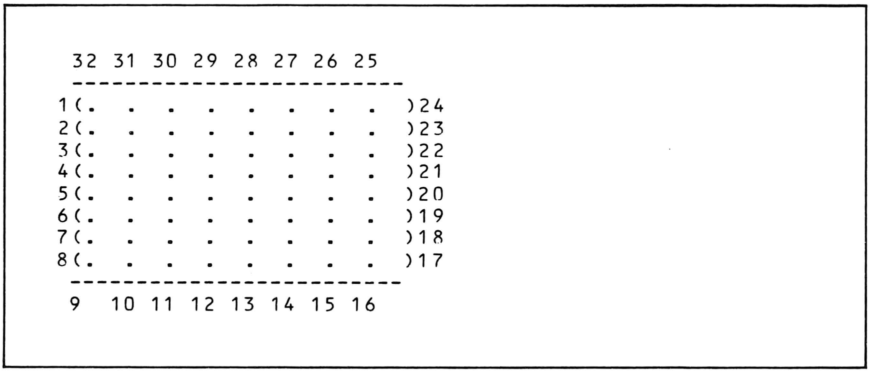
```
ABSORBED
```

or

```
IT CAME OUT AT —
```

with a number from 1 to 32 indicating where the ray left the box. From this information you must deduce the location of the balls.

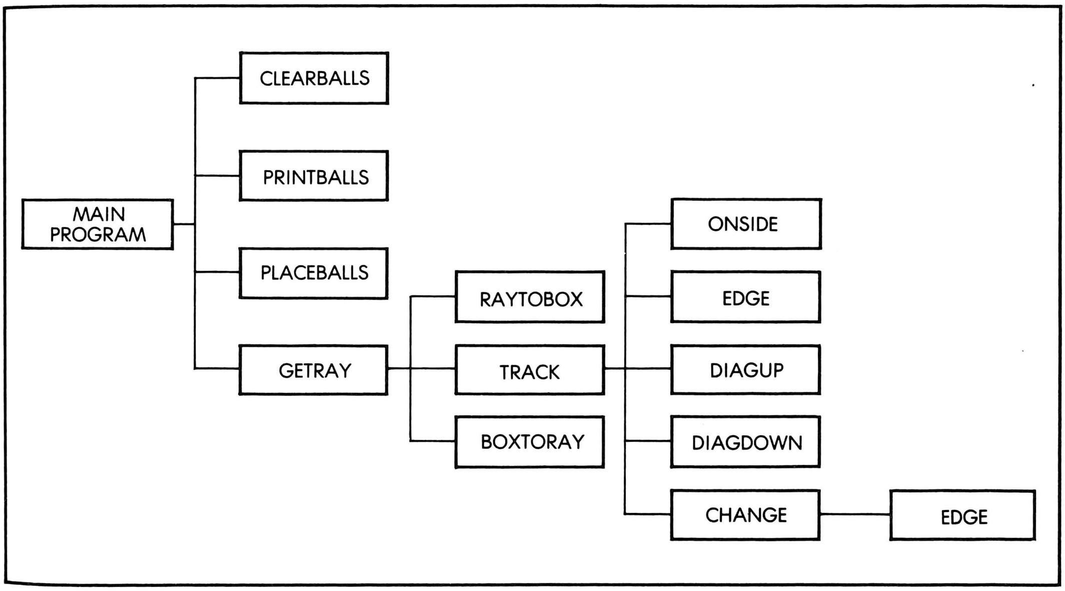
Try to use a minimum number of ray shots to find the balls. When you are ready to compare your guess to the actual location of the balls, type a 0 in response to the prompt. The entire box will be displayed for you, with the balls (represented by asterisks) in place.



```
  32 31 30 29 28 27 26 25
  -------------------------
 1(.  .  .  .  .  .  .  .  )24
 2(.  .  .  .  .  .  .  .  )23
 3(.  .  .  .  .  .  .  .  )22
 4(.  .  .  .  .  .  .  .  )21
 5(.  .  .  .  .  .  .  .  )20
 6(.  .  .  .  .  .  .  .  )19
 7(.  .  .  .  .  .  .  .  )18
 8(.  .  .  .  .  .  .  .  )17
  -------------------------
  9  10 11 12 13 14 15 16
```

```
HOW MANY BALLS? 4
WHERE SHOULD I SHOOT NOW? 7
IN AT  7 ABSORBED
WHERE SHOULD I SHOOT NOW? 21
IN AT 21 IT CAME OUT AT 11
WHERE SHOULD I SHOOT NOW? 31
IN AT 31 IT CAME OUT AT 28
WHERE SHOULD I SHOOT NOW? 12
IN AT 12 IT CAME OUT AT 29
WHERE SHOULD I SHOOT NOW? 10
IN AT 10 ABSORBED
WHERE SHOULD I SHOOT NOW? 0
  32 31 30 29 28 27 26 25
  -------------------------
 1(.  .  .  .  .  .  .  .  )24
 2(.  .  .  .  .  .  .  .  )23
 3(*  .  .  .  .  *  .  .  )22
 4(.  .  .  .  .  .  .  .  )21
 5(.  *  .  .  .  .  .  .  )20
 6(.  .  .  .  .  .  .  .  )19
 7(.  .  .  .  .  .  .  *  )18
 8(.  .  .  .  .  .  .  .  )17
  -------------------------
  9  10 11 12 13 14 15 16
```

# The Program.


CLEARBALLS
PRINTBALLS
MAIN PROGRAM
PLACEBALLS
GETRAY
RAYTOBOX
TRACK
BOXTORAY
ONSIDE
EDGE
DIAGUP
DIAGDOWN
CHANGE
EDGE

The main data structure of this program is BOX, which is an array of arrays of BOOLEAN elements. A TRUE value for any one of these elements will indicate the location of a ball in the grid.

The main program is simply a series of five procedure calls.

Procedure CLEARBALLS initializes all of the elements of BOX to FALSE. Procedure PRINTBALLS is called twice during the game to display the box. At the beginning of the program (before the balls have been placed) each compartment of the grid is represented by a period (.). At the end of the program the box is printed again to display the position of the balls. Notice that a reference to an element of BOX represents a BOOLEAN value in the **IF** statement:

```
IF BOX[I][J] THEN
   WRITE ('* ')
```

Procedure PLACEBALLS prompts the user to input the number of balls, and makes two calls to RANDOM to determine the position coordinates of each ball. Subsequently, that position of BOX is set to TRUE, to represent a ball:

```
BOX[I][J] := TRUE;
```

Procedure GETRAY asks the user to input the starting point of a ray. After validation, the user's input is converted to a number from 1 to 64 (representing one of the 64 compartments of the grid) by procedure RAYTOBOX. RAYTOBOX also sets the variable DIRECTION to its correct value (RIGHT, UP, LEFT, or DOWN). GETRAY then calls TRACK to determine what happens when the ray enters the box.

Function TRACK stores the source location of the ray (a number from 1 to 64) in the formal parameter PLACE. The main block of TRACK thus begins with two formulas that convert PLACE into the row number and column number of the ray's starting point:

```
ROWNUM := (PLACE - 1) DIV SIDE + 1;
COLNUM := (PLACE - 1) MOD SIDE + 1;
```

(The best way to convince yourself that these formulas work is to write out the 64 compartment numbers of the 8-by-8 grid and study the relationship between the outer perimeter of numbers and their corresponding row and column numbers. Later we will see that

TRACK is called recursively to trace the path of the ray through the grid; thus these formulas are used to find the row and column numbers of *any* number in the grid.)

If the ray has entered a compartment that contains a ball, then TRACK returns a 0 to indicate that the ray was absorbed:

```
IF BOX[ROWNUM][COLNUM] THEN
   TRACK := ABSORBED
```

Otherwise, in a series of nested **IF/THEN/ELSE** statements, TRACK calls the functions ONSIDE, DIAGUP, and DIAGDOWN to test for the location of any balls that might be on or near the ray's path. These functions all work by incrementing or decrementing the row and column numbers and returning the value (TRUE or FALSE) of the resulting position of BOX. ONSIDE can only be TRUE when the ray is first sent out—if there is a ball on the edge of the box, adjacent to the source of the ray. DIAGUP and DIAGDOWN use **CASE** statements to test for all possible combinations of balls located on a diagonal to the ray's present position.

If any of these functions (or a combination of these functions, in the case of DIAGUP **AND** DIAGDOWN) return a TRUE value, then the DIRECTION of the ray must be adjusted accordingly. This is done by **CASE** statements inside TRACK. Once DIRECTION has been reset, TRACK calls CHANGE to "increment" the position of the ray. (CHANGE calls EDGE to make sure the ray has not yet reached the edge of the box.) Then TRACK is called recursively to continue the ray's trip through the box.

The structure of the **IF/THEN/ELSE** statements ensures that TRACK will end either when the ray has been absorbed or when it has travelled all the way to a new edge of the box. TRACK returns the final location of the ray (an integer from 1 to 64, or a 0 for an absorbed ray) to GETRAY. GETRAY then calls BOXTORAY to convert that number to an entry/exit number (from 1 to 32) to display to the player.

When the player enters a 0 for RAYNUM, control returns to the main program, which calls PRINTBALLS again, this time to display the correct position of the balls.

```
(*
 * BLACKBOX
 * TRY TO DETERMINE THE LOCATION OF
 * N BALLS IN A BLACK BOX
 *)
PROGRAM BLACKBOX(INPUT,OUTPUT);
USES APPLESTUFF;

CONST
   SIDE   =  8;                 (* THERE ARE 8 SQUARES TO A SIDE *)
   SIDEN1 =  9;                 (* THE LENGTH OF ONE SIDE +1 *)
   MAXSTART = 32;               (* MAXIMUM POSSIBLE LOCATION FOR RAY *)
   ABSORBED =  0;               (* WHAT TRACK RETURNS WHEN RAY ABSORBED *)
   BOXSIZE  = 64;               (* SIZE OF THE BOX *)

TYPE
   COURSE   = (UP,DOWN,LEFT,RIGHT);     (* POSSIBLE DIRECTIONS FOR RAYS *)

   LINES    = ARRAY[0..SIDEN1] OF BOOLEAN; (* POSSIBLE LOCATIONS IN THE BOX
   LOCATION= 0..BOXSIZE;                    * 0 IS FOR ABSORPTION
                                            *)


VAR
   BOX    : ARRAY [0..SIDEN1] OF LINES;   (* THE BLACK BOX *)
   DIRECTION : COURSE;                  (* THE RAY'S CURRENT DIRECTION *)

   STARTRAY  : 1..MAXSTART;             (* THE RAY'S STARTING LOCATION *)

   NUMBALLS  : INTEGER;                 (* NUMBER OF BALLS *)


   (*
    * TRACK
    * RECURSIVE FUNCTION THAT FOLLOWS
    * A RAY'S COURSE THROUGH THE BOX
    *)
   FUNCTION TRACK(PLACE : LOCATION):LOCATION;
   VAR
      ROWNUM : 1..SIDE;         (* FIRST INDEX INTO BOX ARRAY *)
      COLNUM : 1..SIDE;         (* SECOND INDEX INTO BOX ARRAY *)

      (*
       * ONSIDE
       * TESTS TO SEE IF THERE IS A BALL
       * ON EITHER SIDE OF A RAY
       *)
      FUNCTION ONSIDE:BOOLEAN;
      BEGIN
         IF (DIRECTION = RIGHT) OR (DIRECTION = LEFT) THEN
            ONSIDE := BOX[ROWNUM-1][COLNUM] OR BOX[ROWNUM+1][COLNUM]
         ELSE
            ONSIDE := BOX[ROWNUM][COLNUM -1] OR BOX[ROWNUM][COLNUM +1]
      END; (* ONSIDE *)
```

```
(*
 * EDGE
 * TESTS IF A RAY HAS REACHED THE END
 * OF THE BOX
 *)
FUNCTION EDGE:BOOLEAN;
BEGIN
   CASE DIRECTION OF
      LEFT : EDGE := ( (PLACE -1) MOD SIDE ) = 0;
      RIGHT: EDGE := ( (PLACE -1) MOD SIDE ) = SIDE -1;
      UP   : EDGE := ( (PLACE -1) DIV SIDE ) = 0;
      DOWN : EDGE := ( (PLACE -1) DIV SIDE ) = SIDE -1
   END
END; (* EDGE *)

(*
 * DIAGUP
 * TELLS ME IF THERE IS A BALL ON THE
 * UPWARD DIAGONAL
 *)
FUNCTION DIAGUP : BOOLEAN;
BEGIN
   CASE DIRECTION OF
      LEFT  : DIAGUP := BOX[ROWNUM-1][COLNUM-1];
      RIGHT : DIAGUP := BOX[ROWNUM-1][COLNUM+1];
      UP    : DIAGUP := BOX[ROWNUM-1][COLNUM-1];
      DOWN  : DIAGUP := BOX[ROWNUM+1][COLNUM-1]
   END
END; (* DIAGUP *)

(*
 * DIAGDOWN
 * TELLS ME IF THERE
 * IS A BALL ON THE DIAGONAL
 * DOWNWARD
 *)
FUNCTION DIAGDOWN : BOOLEAN;
BEGIN
   CASE DIRECTION OF
      LEFT  : DIAGDOWN := BOX[ROWNUM+1][COLNUM-1];
      RIGHT : DIAGDOWN := BOX[ROWNUM+1][COLNUM+1];
      UP    : DIAGDOWN := BOX[ROWNUM-1][COLNUM+1];
      DOWN  : DIAGDOWN := BOX[ROWNUM+1][COLNUM+1]
   END
END; (* DIAGUP *)

(*
 * CHANGE
 * MOVES THE RAY ONE SQUARE IN THE
 * CURRENT DIRECTION
 *)
```

```
   PROCEDURE CHANGE(VAR NUM:LOCATION);
   BEGIN
      (*
       * IF WE ARE ON AN EDGE, THEN DON'T
       * MOVE
       *)
      IF NOT EDGE THEN
         CASE DIRECTION OF
            LEFT : NUM := NUM -1;

            RIGHT: NUM := NUM +1;

            UP   : NUM := NUM - SIDE;

            DOWN : NUM := NUM +SIDE
         END
   END; (* CHANGE *)

(*
 * TRACK
 *)
BEGIN
   ROWNUM := (PLACE -1) DIV SIDE +1;
   COLNUM := (PLACE -1) MOD SIDE +1;

   IF BOX[ROWNUM][COLNUM] THEN
      TRACK := ABSORBED
   ELSE
   BEGIN
      IF ONSIDE THEN
      BEGIN
            CASE DIRECTION OF
               LEFT : DIRECTION := RIGHT;
               RIGHT: DIRECTION := LEFT;
               UP   : DIRECTION := DOWN;
               DOWN : DIRECTION := UP
            END;
         TRACK := PLACE
      END
      ELSE
      BEGIN
         IF DIAGUP AND DIAGDOWN THEN
         BEGIN
            CASE DIRECTION OF
               LEFT : DIRECTION := RIGHT;
               RIGHT: DIRECTION := LEFT;
               UP   : DIRECTION := DOWN;
               DOWN : DIRECTION := UP
            END;
            CHANGE(PLACE);
            TRACK := TRACK(PLACE)
         END
         ELSE
         BEGIN
            IF DIAGUP THEN
            BEGIN
               CASE DIRECTION OF
                  LEFT, RIGHT:
                     DIRECTION := DOWN;

                  UP, DOWN:
                     DIRECTION := RIGHT
               END;
               CHANGE(PLACE);
               TRACK := TRACK(PLACE)
            END
            ELSE
            IF DIAGDOWN THEN
            BEGIN
               CASE DIRECTION OF
                  LEFT,RIGHT :
                     DIRECTION := UP;

                  UP, DOWN :
                     DIRECTION := LEFT
               END;
               CHANGE(PLACE);
               TRACK := TRACK(PLACE)
            END
            ELSE
            IF NOT EDGE THEN
            BEGIN
               CHANGE(PLACE);
               TRACK := TRACK(PLACE)
            END
            ELSE
               TRACK := PLACE
         END
      END
   END

END; (* TRACK *)
(*
 * CLEARBALLS
 * REMOVES ALL THE BALLS FROM THE BOX
 *)
PROCEDURE CLEARBALLS;
VAR
   I,J : 0..SIDEN1;          (* INDEXES *)
BEGIN
   FOR I := 0 TO SIDEN1 DO
      FOR J := 0 TO SIDEN1 DO
         BOX[I][J] := FALSE
END; (* CLEARBALLS *)

(*
 * PLACEBALLS
 * ASKS THE USER FOR THE NUMBER OF BALLS
 * TO PLACE IN THE BOX, THEN RANDOMLY
 * PLACES THE BALLS IN THE BOX
 *)
```

```
PROCEDURE PLACEBALLS;
VAR
   I,J : 0..SIDEN1;            (* INDEXES USED TO CLEAR BOX *)
   PLACED : INTEGER;           (* THE NUMBER OF BALLS SO FAR PLACED *)
BEGIN
   RANDOMIZE;
   CLEARBALLS;                 (* REMOVE OLD BALLS *)

   WRITELN;
   WRITE('HOW MANY BALLS? ');
   READLN(NUMBALLS);
   RANDOMIZE;
   PLACED := 0;
   WHILE PLACED < NUMBALLS DO
   BEGIN
      I := 1 + RANDOM MOD SIDE;
      J := 1 + RANDOM MOD SIDE;
      IF NOT BOX[I][J] THEN
      BEGIN
         BOX[I][J] := TRUE;
         PLACED := PLACED +1
      END
   END
END; (* PLACEBALLS *)

(*
 * GETRAY
 * SITS IN A LOOP GETTING NUMBERS FROM THE USER,
 * AND USING THEM TO DETERMINE WHERE TO FIRE A RAY.
 * IF THE NUMBER IS 0, THEN THE USER IS READY
 * TO GUESS WHERE THE BALLS ARE LOCATED
 *)
PROCEDURE GETRAY;
VAR
   RAYNUM : INTEGER;           (* NUMBER TYPED IN BY USER *)
   OUTAT  : LOCATION;          (* WHERE THE RAY COMES OUT *)

   (*
    * RAYTOBOX
    * CONVERTS A USER NUMBER INTO THE
    * EQUIVALENT LOCATION IN  THE BOX
    *)
   PROCEDURE RAYTOBOX(VAR NUM : LOCATION);
   BEGIN
      IF NUM <> 0 THEN
      BEGIN
         IF NUM <= SIDE THEN
         BEGIN
            DIRECTION := RIGHT;
            NUM := (NUM -1)*SIDE +1
         END
         ELSE
         IF NUM <= (SIDE + SIDE) THEN
         BEGIN
            DIRECTION := UP;
            NUM := SQR(SIDE) - ((2*SIDE)-NUM)
         END
```

```
           ELSE
           IF NUM <= (3*SIDE) THEN
           BEGIN
              DIRECTION := LEFT;
              NUM := (3*SIDE -NUM +1)*SIDE
           END
           ELSE
           BEGIN
              DIRECTION := DOWN;
              NUM := (4*SIDE -NUM +1)
           END
        END
     END; (* RAYTOBOX *)

     (*
      * BOXTORAY
      * RETURNS THE RAY LOCATION OF AN EDGE
      *)
     FUNCTION BOXTORAY(PLACE:LOCATION):LOCATION;
     BEGIN
        CASE DIRECTION OF
           RIGHT : BOXTORAY := 25 - PLACE DIV SIDE;
           LEFT  : BOXTORAY := (PLACE + (SIDE-1)) DIV SIDE;
           UP    : BOXTORAY := 33 - PLACE;
           DOWN  : BOXTORAY := PLACE - 48
        END
     END; (* BOXTORAY *)

  BEGIN
     REPEAT
        WRITE('WHERE SHOULD I SHOOT NOW? ');
        READLN(RAYNUM);
        WHILE NOT (RAYNUM IN [0..MAXSTART]) DO
        BEGIN
           WRITELN('SORRY, BUT ONLY NUMBERS BETWEEN 0 AND ',MAXSTART);
           WRITE('START WHERE? ');
           READLN(RAYNUM)
        END;
        IF RAYNUM <> 0 THEN
        BEGIN
           WRITE('IN AT ',RAYNUM:2,' ');
           OUTAT := 0;
           RAYTOBOX(RAYNUM);
           OUTAT := TRACK(RAYNUM);
           IF OUTAT = 0 THEN
              WRITELN('ABSORBED')
           ELSE
              WRITELN('IT CAME OUT AT ',BOXTORAY(OUTAT))
        END

     UNTIL RAYNUM = 0
  END; (* GETRAY *)

  (*
   * PRINTBALLS
   * PRINTS OUT THE BALLS
   *)
   PROCEDURE PRINTBALLS;
   VAR
      I,J : 1..SIDE;            (* INDEXES INTO THE BOX *)
   BEGIN

      WRITELN('  32 31 30 29 28 27 26 25');
      WRITELN('  -------------------------');
      FOR I := 1 TO SIDE DO
      BEGIN
         WRITE(I:2,'(');
         FOR J := 1 TO SIDE DO
            IF BOX[I][J] THEN
               WRITE('*  ')
            ELSE
               WRITE('.  ');
         WRITELN(')',25-I)
      END;
      WRITELN('  -------------------------');
      WRITELN('  9  10 11 12 13 14 15 16')
   END; (* PRINTBALLS *)
BEGIN
   CLEARBALLS;
   PRINTBALLS;
   PLACEBALLS;
   GETRAY;
   PRINTBALLS;

END.
```

# TELEPHONE

## The Game.

For anyone who has trouble remembering important phone numbers, this is a program that can truly make life a little easier. While not precisely a game, Telephone can produce some amusing, yet useful, results. It lists all the pronounceable "words" that can be associated with a given number.

You will be asked to enter a telephone number:

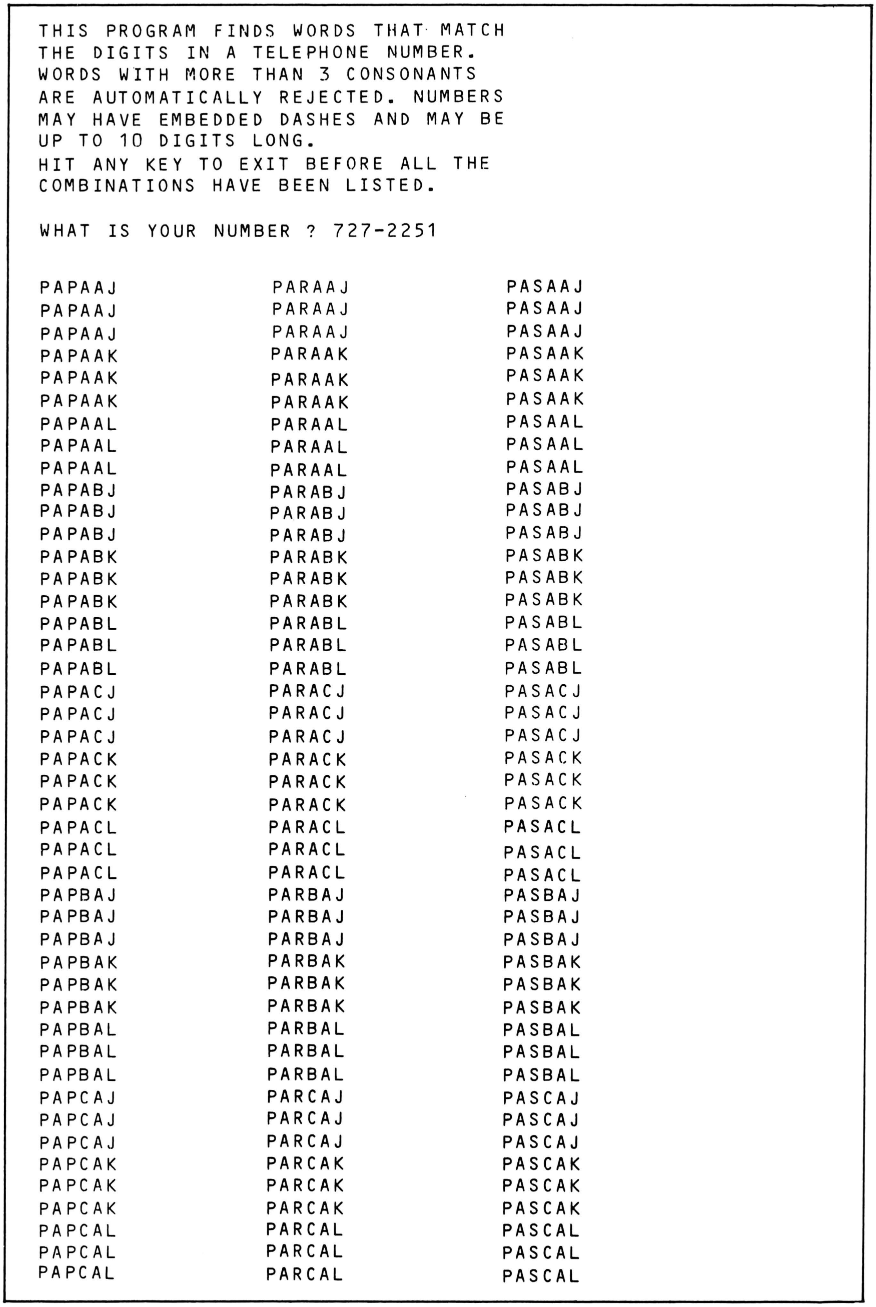
```
WHAT IS YOUR NUMBER?
```

You may type the number with or without hyphens (though no other character will be accepted). The computer will begin listing the word combinations that can be produced from the three-letter codes associated with the digits 2 to 9 on the telephone dial. (Since 1 and 0 have no letters, a space will be placed in the words wherever those digits appear in your phone number.)

To save you the trouble of sorting through unusable combinations, the program eliminates words that it deems too hard to pronounce. The criterion it uses to make this judgment is this: any word that contains more than three consonants in a row is unpronounceable.

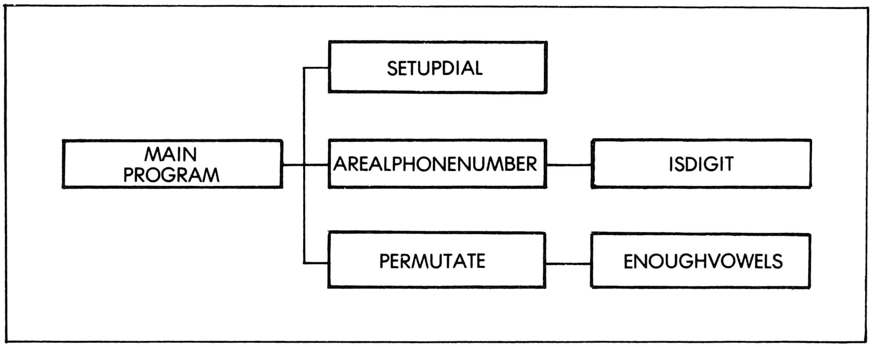
Finally, since this list of words will be extremely long, you may stop the program—whenever you have found a word you like—simply by pressing any key on the keyboard.

TELEPHONE will supply mnemonics for any number up to ten digits long. This allows you to enter a seven-digit number plus its area code, for a complete long-distance number. Now you *really* have no excuse for not giving Mom a call!



```
THIS PROGRAM FINDS WORDS THAT MATCH
THE DIGITS IN A TELEPHONE NUMBER.
WORDS WITH MORE THAN 3 CONSONANTS
ARE AUTOMATICALLY REJECTED. NUMBERS
MAY HAVE EMBEDDED DASHES AND MAY BE
UP TO 10 DIGITS LONG.
HIT ANY KEY TO EXIT BEFORE ALL THE
COMBINATIONS HAVE BEEN LISTED.

WHAT IS YOUR NUMBER ? 727-2251


PAPAAJ          PARAAJ          PASAAJ
PAPAAJ          PARAAJ          PASAAJ
PAPAAJ          PARAAJ          PASAAJ
PAPAAK          PARAAK          PASAAK
PAPAAK          PARAAK          PASAAK
PAPAAK          PARAAK          PASAAK
PAPAAL          PARAAL          PASAAL
PAPAAL          PARAAL          PASAAL
PAPAAL          PARAAL          PASAAL
PAPABJ          PARABJ          PASABJ
PAPABJ          PARABJ          PASABJ
PAPABJ          PARABJ          PASABJ
PAPABK          PARABK          PASABK
PAPABK          PARABK          PASABK
PAPABK          PARABK          PASABK
PAPABL          PARABL          PASABL
PAPABL          PARABL          PASABL
PAPABL          PARABL          PASABL
PAPACJ          PARACJ          PASACJ
PAPACJ          PARACJ          PASACJ
PAPACJ          PARACJ          PASACJ
PAPACK          PARACK          PASACK
PAPACK          PARACK          PASACK
PAPACK          PARACK          PASACK
PAPACL          PARACL          PASACL
PAPACL          PARACL          PASACL
PAPACL          PARACL          PASACL
PAPBAJ          PARBAJ          PASBAJ
PAPBAJ          PARBAJ          PASBAJ
PAPBAJ          PARBAJ          PASBAJ
PAPBAK          PARBAK          PASBAK
PAPBAK          PARBAK          PASBAK
PAPBAK          PARBAK          PASBAK
PAPBAL          PARBAL          PASBAL
PAPBAL          PARBAL          PASBAL
PAPBAL          PARBAL          PASBAL
PAPCAJ          PARCAJ          PASCAJ
PAPCAJ          PARCAJ          PASCAJ
PAPCAJ          PARCAJ          PASCAJ
PAPCAK          PARCAK          PASCAK
PAPCAK          PARCAK          PASCAK
PAPCAK          PARCAK          PASCAK
PAPCAL          PARCAL          PASCAL
PAPCAL          PARCAL          PASCAL
PAPCAL          PARCAL          PASCAL
```

# The Program.



The main program calls procedure SETUPDIAL to initialize the array DIAL. DIAL is an array of arrays of characters, which will hold the three letters that go with each digit.

SETUPDIAL first initializes values for 0 and 1, the two digits that have no letters on the dial. A pair of nested **FOR** loops is used to set the elements for the digits to blank:

```
DIAL[DIGIT][I] := ' ';
```

The next pair of **FOR** loops initializes the elements for the rest of the digits. LETTER is initialized to A; then the SUCC function is used to "increment" LETTER for each succeeding assignment:

```
DIAL[DIGIT][I] := LETTER;
LETTER := SUCC(LETTER);
```

Notice that Q is skipped, since there is no Q on the telephone dial:

```
IF LETTER = 'Q' THEN
   LETTER := SUCC(LETTER);
```

Next in the main program, the set VOWELS is assigned (to be used in function ENOUGHVOWELS):

```
VOWELS := ['A', 'E', 'I', 'O', 'U', 'Y'];
```

A series of instruction lines are written, a prompt is displayed for the input, and the phone number is read into the variable PHONENUM. A **REPEAT/UNTIL** loop calls function AREALPHONENUMER to validate the input.

AREALPHONENUMBER actually performs three tasks. It deletes any hyphens that are embedded in the number; it checks the length of the number to make sure it is within the acceptable range of digits; and, once the hyphens have been removed, it makes sure the number contains only digits.

The hyphens are removed via the UCSD function POS and procedure DELETE. POS has two string parameters; it returns an integer that represents the *position* of the first string in the second string. (POS returns a 0 if the first string does *not* occur in the second string.) DELETE takes three parameters: a string, and two integers. (The string is a **VAR** parameter.) The first integer indicates a position in the string; the second integer indicates how many characters are to be deleted. Thus, the statement:

```
WHILE POS('-',PHONENUM)< >0 DO
   DELETE(PHONENUM,POS('-',PHONENUM),1);
```

searches for and deletes hyphens until POS returns a 0, indicating that all the hyphens have been removed.

Next the length of the phone number (found through a call to the LENGTH function) is validated. Finally, each digit of the phone number is sent to function ISDIGIT. ISDIGIT merely checks to see if the character it receives is between '0' and '9'. If one of the digits is invalid, then AREALPHONENUMBER is ended by the EXIT procedure.

If AREALPHONENUMBER returns a TRUE, then the main program calls procedure PERMUTATE to initiate the main action of the program:

```
PERMUTATE(1);
```

The parameter of this call (1) represents the position of the first digit of the number. This parameter value will be incremented in subsequent recursive calls to procedure PERMUTATE.

PERMUTATE's first statement contains a call to the function KEYPRESS. KEYPRESS returns a TRUE if any key has been pressed since the last READ statement. If a TRUE is returned (meaning the user has seen enough words), the program is terminated:

```
IF KEYPRESS THEN
   EXIT(TELEPHONE);
```

The next **IF/THEN/ELSE** statement in PERMUTATE is a fine demonstration of the power of recursion in Pascal. The statement that assigns values to the array WORD, and the recursive call to PERMUTATE are both contained inside a **FOR** loop, which assures that every permutation of letter combinations will be covered.

Function ENOUGHVOWELS is called before each recursive call to PERMUTATE. If the letter of the word is **IN** the set VOWELS, then the position of the letter is recorded:

```
IF WORD[POSINWORD] IN VOWELS THEN
   LASTVOWEL := POSINWORD
```

ENOUGHVOWELS returns a FALSE if the present position of the word (POSINWORD) is more than three positions away from the last vowel (LASTVOWEL).

If ENOUGHVOWELS returns a FALSE to PERMUTATE, then the current permutation is scrapped; otherwise, the recursive call is executed:

```
IF ENOUGHVOWELS(POSITION) THEN
   PERMUTATE(POSITION + 1);
```

```
PROGRAM TELEPHONE;
USES
   APPLESTUFF;           (* FOR KEYPRESS FUNCTION *)
CONST
    LETTERSPERDIGIT = 3;        (* EACH NUMBER CAN HAVE 3 LETTERS *)
    MAXNUMLENGTH    = 10;       (* ALLOWS LONG-DISTANCE NUMBERS   *)
    MAXCONSONANTS   = 3;        (* MAXIMUM NUMBER OF CONSONANTS   *)
                                (* THAT ARE ALLOWED IN A WORD, IF *)
                                (* THERE ARE MORE THAN THIS MANY  *)
                                (* THE REST OF THE COMBINATIONS   *)
                                (* WITH THIS ROOT ARE SKIPPED     *)
TYPE
    DIGITSLET = ARRAY[1 .. LETTERSPERDIGIT] OF CHAR;
VAR
   DIAL : ARRAY['0' .. '9'] OF DIGITSLET;
   VOWELS : SET OF 'A' .. 'Z';
   PHONENUM : STRING;
   WORD : PACKED ARRAY [1 .. MAXNUMLENGTH] OF CHAR;
   NUMLENGTH : INTEGER;
(*
 * INITIALIZES THE ARRAY DIAL TO CONTAIN
 * THE SAME (DIGIT, 3 LETTERS) GROUPS THAT
 * ARE ON TELEPHONES.  0 AND 1 DO DO NOT HAVE
 * ANY, AND SO ARE SET TO BLANK.  THE REST
 * OF THE NUMBERS ARE MAPPED TO SUCCEEDING
 * TRIPLES OF LETTERS ( 2 -> A,B,C ).  THE
 * EXCEPTION TO THIS IS THAT THE LETTER Q
 * IS IGNORED, MAKING THE TRIPLE FOR 7 TO
 * BE P,R,S.
 *)

PROCEDURE SETUPDIAL;
VAR
   I        : INTEGER;
   DIGIT    : CHAR;
   LETTER   : CHAR;
BEGIN
   FOR DIGIT := '0' TO '1' DO           (* THESE DIGITS HAVE NO LETTERS *)
      FOR I := 1 TO LETTERSPERDIGIT DO  (* ON TELEPHONE DIALS           *)
         DIAL[DIGIT][I] := ' ';
   LETTER := 'A';
   FOR DIGIT := '2' TO '9' DO
      FOR I := 1 TO LETTERSPERDIGIT DO
         BEGIN
         IF LETTER = 'Q' THEN          (* Q IS NOT ON THE DIAL, SKIP TO R *)
            LETTER := SUCC(LETTER);
         DIAL[DIGIT][I] := LETTER;
         LETTER := SUCC(LETTER);
      END;
END; (* SETUPDIAL *)

(*
 * ENOUGHVOWELS APPLIES A SIMPLE TEST TO THE
 * WORD TO FIND OUT IF IS IS PRONOUNCEABLE
 *)
```

```
FUNCTION ENOUGHVOWELS(SOFAR : INTEGER) : BOOLEAN;
VAR
   POSINWORD, LASTVOWEL : INTEGER;
BEGIN
   LASTVOWEL := 0;
   IF SOFAR > MAXCONSONANTS THEN        (* DON'T BOTHER CHECKING FOR    *)
      FOR POSINWORD := 1 TO SOFAR DO    (* TOO MANY CONSONANTS IF THERE *)
      BEGIN                             (* AREN'T THAT MANY LETTERS     *)
         IF WORD[POSINWORD] IN VOWELS THEN
            LASTVOWEL := POSINWORD
         ELSE IF POSINWORD - LASTVOWEL > MAXCONSONANTS THEN
         BEGIN
            ENOUGHVOWELS := FALSE;
            EXIT(ENOUGHVOWELS);
         END;
      END;
   ENOUGHVOWELS := TRUE;
END; (* ENOUGHVOWELS *)

(*
 * THIS IS A RECURSIVE PROCEDURE WHICH PRINTS ALL
 * POSSIBLE COMBINATIONS OF LETTERS FOR THE DIGITS
 * IN THE GIVEN PHONE NUMBER.  GIVEN A POSITION IN
 * THE NUMBER AT WHICH TO CONTINUE LOOKING, IT LOOPS
 * THROUGH ALL THE POSSIBLE VALUES OF THE CURRENT
 * POSITION'S LETTER ( DETERMINED BY THE CORRESPONDING
 * LETTERS FOR THAT DIGIT ON A TELEPHONE DIAL).  FOR
 * EACH LETTER, IT CALLS ITSELF RECURSIVELY ON THE NEXT
 * POSITION IN THE NUMBER (TO FIGURE OUT THE NEXT LETTER).
 * THE END OF THE NUMBER IS REACHED WHEN THE POSITION
 * WOULD EXTEND PAST THE END OF THE STRING; THEN THE
 * CURRENT PERMUTATION IS PRINTED AND IT RETURNS TO GET
 * NEXT COMBINATION.
 *)

PROCEDURE PERMUTATE(POSITION : INTEGER);
VAR
   I : INTEGER;
BEGIN
   IF KEYPRESS THEN             (* USER HITS ANY KEY TO LEAVE          *)
      EXIT(TELEPHONE);          (* BEFORE ALL PERMUTATIONS ARE DONE    *)
   IF POSITION > NUMLENGTH THEN         (* HAVE DONE ALL OF THE DIGITS *)
      WRITELN(WORD:NUMLENGTH)
   ELSE                                 (* GO STILL FURTHER INTO NUMBER  *)
      FOR I := 1 TO LETTERSPERDIGIT DO
      BEGIN
         WORD[POSITION] := DIAL[PHONENUM[POSITION]][I];
         IF ENOUGHVOWELS(POSITION) THEN(*SKIP WORD IF PORTION SO FAR COULD*)
            PERMUTATE(POSITION + 1);   (*NOT POSSIBLY BE PRONOUNCEABLE     *)
      END;
END;

(*
 * RETURNS TRUE IF CH IS A NUMERICAL CHARACTER
 *)
```

```
FUNCTION ISDIGIT (CH : CHAR) : BOOLEAN;
BEGIN
   ISDIGIT := (CH >= '0') AND (CH <= '9');
END;

(*
 * DETERMINES IF THE STRING TYPED IN MIGHT
 * BE A SYNTACTICALLY CORRECT TELEPHONE
 * NUMBER.  IT MUST CONTAIN DIGITS; IT MAY
 * OPTIONALLY HAVE DASHES ( WHICH ARE REMOVED )
 *)

FUNCTION AREALPHONENUMBER : BOOLEAN;
VAR
   DIGITPOS : INTEGER;
BEGIN
   WHILE POS('-', PHONENUM) <> 0 DO     (* GET RID OF ANY DASHES THAT MAY *)
      DELETE(PHONENUM, POS('-', PHONENUM),1);(*SEPARATE PARTS OF THE NUMBER*)
   NUMLENGTH := LENGTH(PHONENUM);
   IF (NUMLENGTH <= 0) OR (NUMLENGTH > MAXNUMLENGTH) THEN
      AREALPHONENUMBER := FALSE
   ELSE
      FOR DIGITPOS := 1 TO NUMLENGTH DO
         IF NOT ISDIGIT(PHONENUM[DIGITPOS]) THEN
         BEGIN
            AREALPHONENUMBER := FALSE;
            EXIT(AREALPHONENUMBER);
         END;
   AREALPHONENUMBER := TRUE;
END; (* AREALPHONENUMBER *)

BEGIN           (* MAIN PROGRAM *)
   SETUPDIAL;
   VOWELS := ['A', 'E', 'I', 'O', 'U', 'Y'];
   WRITELN('THIS PROGRAM FINDS WORDS THAT MATCH');
   WRITELN('THE DIGITS IN A TELEPHONE NUMBER.');
   WRITELN('WORDS WITH MORE THAN ', MAXCONSONANTS:1, ' CONSONANTS');
   WRITELN('ARE AUTOMATICALLY REJECTED. NUMBERS');
   WRITELN('MAY HAVE EMBEDDED DASHES AND MAY BE');
   WRITELN('UP TO ', MAXNUMLENGTH:1, ' DIGITS LONG.');
   WRITELN('HIT ANY KEY TO EXIT BEFORE ALL THE');
   WRITELN('COMBINATIONS HAVE BEEN LISTED.');
   WRITELN;
   REPEAT
      WRITE('WHAT IS YOUR NUMBER ? ');
      READLN(PHONENUM);
   UNTIL AREALPHONENUMBER;
   PERMUTATE(1);
END.
```

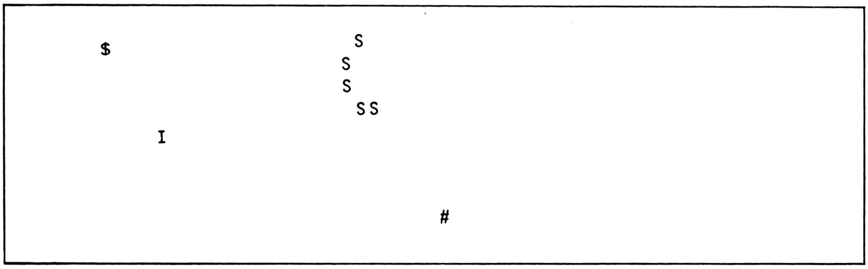
DL

# SNAKE

**The Game.** Come and visit the snake. Despite his reputation, the snake can be a reasonably hospitable fellow. Up to a certain point he seems to enjoy your company—or at least to be willing to ignore you as you scurry about gathering up the pots of money hidden in his domain. But beware; the snake dislikes greedy visitors. If you try to take *too much* of his money he may just eat you before you can escape.

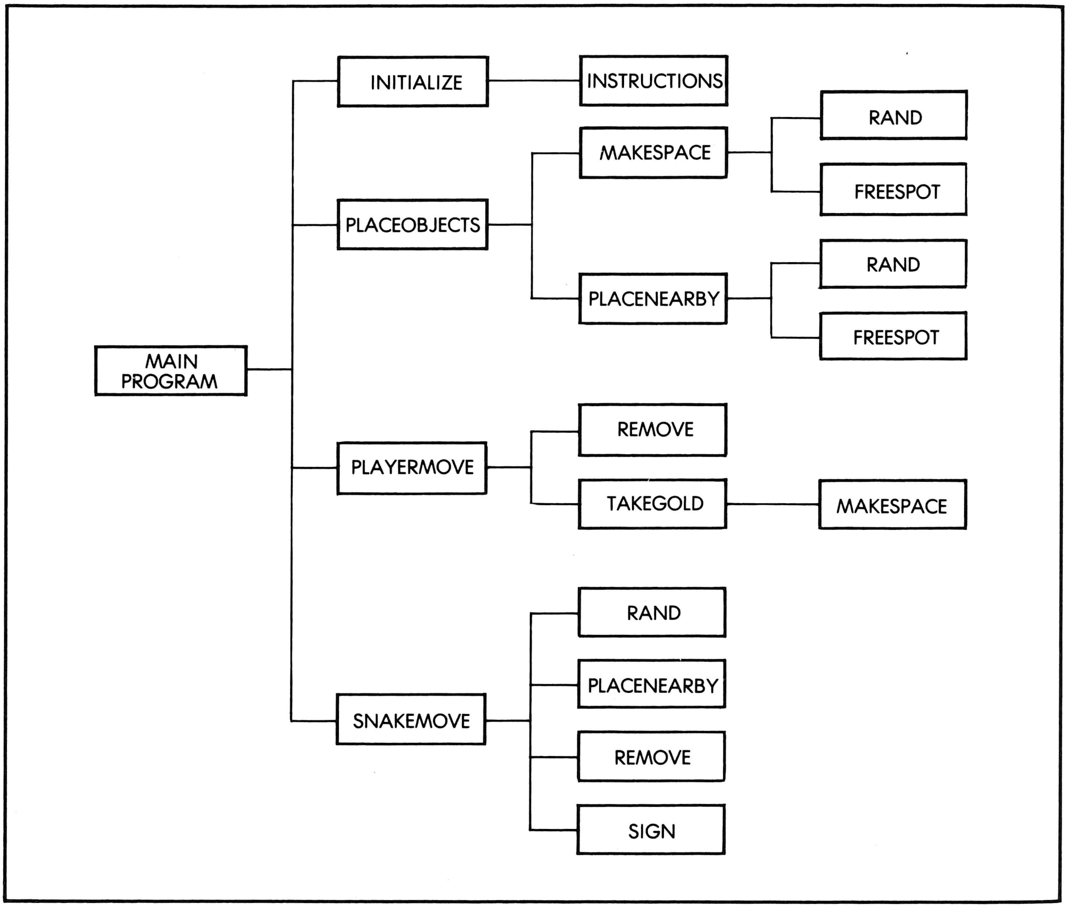
Your position on the screen is represented by the letter I. You can move the I up, down, right, or left by pressing the U, N, H, or J keys, respectively. (Notice that these four keys form a cross on your keyboard.) The dollar signs ($) that appear one by one on the screen represent the pots of money; each one is worth $25, and you can capture the money by moving the I to the position of the $. (Your current earnings are displayed in the upper left corner of the screen.) The sharp sign (#) is the door; you may end the game with your current earnings by moving the I to the # position.

The second way you can end the game is to be eaten by the snake. The snake is represented on the screen by a slithering string of S characters. Until you have earned over $100, the snake will tend to be fairly benign, meandering about the screen with no particular direction. Once you are worth some money, however, the snake will not want to see you leave through the door, and he will start going after you. (The snake, by the way, can move diagonally, which you cannot do.) See how much money you can get out the door, but watch out for the snake; once he warms up to the chase, he develops a hearty appetite.



```
       $                S
                       S
                       S
                        SS
           I



                              #
```

# The Program.


MAIN PROGRAM
INITIALIZE
INSTRUCTIONS
PLACEOBJECTS
MAKESPACE
RAND
FREESPOT
PLACENEARBY
RAND
FREESPOT
PLAYERMOVE
REMOVE
TAKEGOLD
MAKESPACE
SNAKEMOVE
RAND
PLACENEARBY
REMOVE
SIGN

The scalar type THING, as defined in the type declaration section of this program, determines the range of values of the array SCREEN, and indirectly represents the characters of this game:

```
THING = (PLAYERTHING, SNAKETHING, MONEYTHING,
         DOORTHING, EMPTYTHING, SCORETHING);
```

Procedures INITIALIZE and PLACEOBJECTS (called from the main program) set the scene by assigning values of type THING to the elements of SCREEN.

Procedure INITIALIZE begins with a call to procedure INSTRUCTIONS, which offers the reader a description of the game. Then, in a pair of nested **FOR** loops, INITIALIZE sets all of the elements of SCREEN to EMPTYTHING. SCREEN is a two-dimensional array (or, more precisely, an array of arrays):

```
FOR X := 0 TO WIDTH DO
  FOR Y := 0 TO HEIGHT DO
    SCREEN[X][Y] := EMPTYTHING;
```

INITIALIZE then calls RANDOMIZE to set the "seed" for the RANDOM function. (RANDOM will be used for the initial positioning of the characters on the screen, and up to a point, for the meanderings of the snake.) Then beginning values are assigned to several other variables. LOOKSLIKE, an array of CHAR (indexed by values of type THING), holds the actual characters that will be displayed on the screen to represent the snake, the player, the money, and the door. In addition, one element of LOOKSLIKE is set for the blank positions on the screen:

```
LOOKSLIKE[EMPTYTHING] := ' ';
```

LEFT and EATEN are BOOLEAN variables that trigger the end of the game (if the player goes out the door, or if the snake eats the player). Both are initialized to FALSE. SCORE, representing the amount of money the player has captured, starts at 0. Finally, the first ten characters of the top line of the screen are reserved for a display of the score:

```
FOR X := 0 TO 10 DO
  SCREEN[X,0] := SCORETHING;
```

The last line of INITIALIZE clears the screen, in preparation for the play. (The value of CLEARSCREEN is set in the **CONST** section.)

Procedure PLACEOBJECTS makes calls to procedure MAKESPACE for the head of the snake (SNAKE[1]) and for each of the single-character displays on the screen. Variables PLAYER, MONEY, and DOOR are all of type COORDINATE (a record type, with two fields, X and Y); each of them represents the current position of a character. SNAKE is an array of COORDINATE. Thus, the calls to MAKESPACE have two parameters: a variable of type COORDINATE and a value of type THING, representing the position and the category of the character that is to be placed on the screen.

MAKESPACE receives these two values in the formal parameters NEWPOS and FORWHAT. NEWPOS is a **VAR** parameter; this is because MAKESPACE actually initializes the positions of the characters. To do so, it calls two functions: RAND and FREESPOT. RAND simply uses the RANDOM function to produce a number within a given range. FREESPOT checks the random coordinates it is sent, to see if the corresponding position in SCREEN has the value EMPTYTHING. (In other words, FREESPOT returns a TRUE if the position can receive a character.)

Notice the economy of MAKESPACE's random search for a free space for each character:

```
WITH NEWPOS DO
  BEGIN
    REPEAT
      X := RAND(0,WIDTH - 1);
      Y := RAND(0,HEIGHT - 1)
    UNTIL FREESPOT(NEWPOS);
```

The **WITH** statement allows direct reference to the fields (X and Y) of the record NEWPOS. The **REPEAT/UNTIL** loop sends each new set of coordinates to FREESPOT, and continues looping until FREESPOT returns a TRUE.

When a free position is found, the cursor is moved to that position:

```
GOTOXY(X,Y);
```

the appropriate character (found in the array LOOKSLIKE) is written:

```
WRITE(LOOKSLIKE[FORWHAT]);
```

and the position is recorded in the array SCREEN:

```
SCREEN[X,Y] := FORWHAT;
```

Procedure PLACEOBJECTS creates the rest of the snake's body via calls to procedure PLACENEARBY:

```
FOR SNAKEBODY := 2 TO SNAKELENGTH DO
   PLACENEARBY(SNAKE[SNAKEBODY],SNAKE[SNAKEBODY—1]);
```

PLACENEARBY (which is also used later in the program to move the snake) randomly places a new part of the snake's body within the 3-by-3 matrix around the previous part. RAND is called to determine the delta x and delta *y* values between – 1 and 1 (but only one of these values can be set to zero; if both were zero, the new part of the snake would be placed directly on the old part):

```
REPEAT
   DELTAX := RAND(—1,1);
   DELTAY := RAND(—1,1);
UNTIL (DELTAX <> 0) OR (DELTAY <> 0);
NEAR.X := COORD.X + DELTAX;
NEAR.Y := COORD.Y + DELTAY;
```

An outer **REPEAT/UNTIL** loop calls FREESPOT to make sure the position is free; however, a check is also made to see if a position of the snake goes over the player's position (in which case the player is eaten and the game is over):

```
UNTIL (FREESPOT(NEAR)) OR
   (NEAR = PLAYER);
```

Then, just as in MAKESPACE, the cursor is moved, the character is written, and the value is recorded in SCREEN.

Now the game can begin. The main program makes alternate calls to PLAYERMOVE and SNAKEMOVE until the player leaves or is eaten.

Procedure PLAYERMOVE reads the player's direction command from the keyboard, and, in a **CASE** statement, increments or decrements the appropriate coordinate field of PLAYER. (Notice another use of the **WITH** statement.) A call to procedure REMOVE blanks out the player's old position on the screen, and assigns a value of EMPTYTHING to the old position in SCREEN.

If the player lands on some money (**IF** PLAYER = MONEY), procedure TAKEGOLD is called to increment SCORE, to display the new value of the player's assets, and to call MAKESPACE for a new, randomly placed MONEY character. If the player lands on the door (**IF** PLAYER = DOOR), LEFT is set to TRUE, and the game is over. Finally, the player's new position is displayed on the screen and recorded in SCREEN.

At the beginning of the game, and at least until the player has accumulated more than $100, procedure SNAKEMOVE simply calls PLACENEARBY to move the snake randomly about the screen:

```
IF RAND(0,SCORE) <= 100 THEN
   PLACENEARBY(NEWPOS,SNAKE[1])
```

However, when the player has a significant amount of money, the snake starts getting tough. The new moving algorithm directs the snake's head precisely toward the position of the player. Both coordinates are recalculated, so that the snake can move on a diagonal if it needs to:

```
NEWPOS.X := SNAKE[1].X + SIGN(PLAYER.X -SNAKE[1].X);
NEWPOS.Y := SNAKE[1].Y + SIGN(PLAYER.Y - SNAKE[1].Y);
```

The call to function SIGN simply returns a 1 if the snake is above or to the right of the player, and a - 1 if the snake is below or to the left. As long as neither DOORTHING or MONEYTHING is in the way (in which case PLACENEARBY must be called for a random move) the snake's head is moved in this calculated direction. A call to REMOVE near the end of the procedure blanks out the last part of the snake's tail. A **FOR** loop increments the indices to the other elements of SNAKE, and SNAKE[1] is assigned the value of NEWPOS. If the snake moves over the player, EATEN is set to TRUE, and the game is over.

```
PROGRAM SNAKEGAME;

USES
   APPLESTUFF;          (* FOR RANDOM NUMBER FUNCTION *)

CONST
   PLAYERCHAR = 'I';
   SNAKECHAR  = 'S';
   MONEYCHAR  = '$';
   DOORCHAR   = '#';

   UPCOMMAND    = 'U';
   DOWNCOMMAND  = 'N';
   LEFTCOMMAND  = 'H';
   RIGHTCOMMAND = 'J';

   SNAKELENGTH = 5;
   HEIGHT      = 23;    (* SIZE OF THE SCREEN, STARTING AT 0, NOT 1 *)
   WIDTH       = 39;
   CLEARSCREEN = 12;    (* PRINT CHR(12) TO CLEAR THE SCREEN      *)
   MONEYWORTH  = 25;    (* THE SCORE FOR PICKING UP A SINGLE PILE OF MONEY *)



TYPE
   COORDINATE = RECORD  (* A POSITION OF THE SCREEN HAS AN       *)
                  X : INTEGER;  (* X (HORIZONTAL) COLUMN NUMBER  *)
                  Y : INTEGER;  (* AND A Y (VERTICAL) ROW NUMBER *)
                END;
   SNAKETYPE = ARRAY [1 .. SNAKELENGTH]
                  OF COORDINATE;
   THING = (PLAYERTHING, SNAKETHING, MONEYTHING,
            DOORTHING, EMPTYTHING, SCORETHING);

VAR
   SNAKE : SNAKETYPE;
   PLAYER, MONEY, DOOR : COORDINATE;
   SCORE : INTEGER;
   LEFT , EATEN : BOOLEAN;

   SCREEN : ARRAY [0 .. WIDTH]  OF
            ARRAY [0 .. HEIGHT] OF THING;

   LOOKSLIKE : ARRAY [ THING ] OF CHAR; (* WHAT THE VARIOUS OBJECTS LOOK *)
                                        (* LIKE ON THE SCREEN            *)

(*
 * RETURNS A RANDOM INTEGER BETWEEN MIN AND MAX
 *)

FUNCTION RAND(MIN, MAX : INTEGER) : INTEGER;
BEGIN
   RAND := MIN + RANDOM MOD (MAX - MIN +1);
END;
```

```
PROCEDURE INSTRUCTIONS;
VAR
   ANSWER : CHAR;
BEGIN
   WRITE('DO YOU WANT INSTRUCTIONS? ');
   READLN(ANSWER);
   WHILE NOT (ANSWER IN ['Y','N']) DO
   BEGIN
      WRITELN('PLEASE ANSWER YES OR NO.');
      READLN(ANSWER);
   END;
   IF ANSWER = 'Y' THEN
   BEGIN
      WRITELN('THE OBJECT OF SNAKE IS TO GET AS MUCH');
      WRITELN('MONEY TO THE DOOR AS POSSIBLE. THE SNAKE');
      WRITELN('TRIES TO PREVENT YOU.  AS YOU GET MORE');
      WRITELN('MONEY, HE TRIES MORE AND MORE');
      WRITELN('SUCCESSFULLY.  YOU MOVE UP, DOWN, LEFT');
      WRITELN('AND RIGHT BY TYPING U, N, H, AND J');
      WRITELN('RESPECTIVELY.  YOU CANNOT MOVE');
      WRITELN('DIAGONALLY, THOUGH THE SNAKE CAN.');
      WRITE('TYPE RETURN TO CONTINUE :');
      READLN;
   END;
END;

(*
 * SETS UP ALL THE VARIABLES ...
 *)

PROCEDURE INITIALIZE;
VAR
   X, Y : INTEGER;
BEGIN
   INSTRUCTIONS;
   FOR X := 0 TO WIDTH DO
      FOR Y := 0 TO HEIGHT DO
         SCREEN[X][Y] := EMPTYTHING;
   RANDOMIZE;           (* SET RANDOM NUMBER SEED *)
   LOOKSLIKE[SNAKETHING] := SNAKECHAR;
   LOOKSLIKE[PLAYERTHING] := PLAYERCHAR;
   LOOKSLIKE[MONEYTHING] := MONEYCHAR;
   LOOKSLIKE[EMPTYTHING] := ' ';
   LOOKSLIKE[DOORTHING] := DOORCHAR;
   LEFT := FALSE;
   EATEN := FALSE;
   SCORE := 0;
   FOR X := 0 TO 10 DO          (* PREVENTS SCORE FROM BEING *)
      SCREEN[X, 0] := SCORETHING; (* OVERWRITTEN             *)
   WRITE(CHR(CLEARSCREEN));
END;
```

```
(*
 * RETURNS TRUE IF THE POSITION IS VALID AND EMPTY
 *)

FUNCTION FREESPOT(POS : COORDINATE) : BOOLEAN;
BEGIN
   IF (POS.X IN [0 .. WIDTH]) AND (POS.Y IN [0 .. HEIGHT]) THEN
      FREESPOT := SCREEN[POS.X, POS.Y] = EMPTYTHING
   ELSE
      FREESPOT := FALSE;
END;

(*
 * ASSIGNS THE COORDINATES OF A POSITION ON THE SCREEN THAT IS NOT
 * BEING USED
 *)

PROCEDURE MAKESPACE(VAR NEWPOS : COORDINATE; FORWHAT : THING);
BEGIN
   WITH NEWPOS DO
   BEGIN
      REPEAT
         X := RAND(0, WIDTH - 1);
         Y := RAND(0, HEIGHT - 1);
      UNTIL FREESPOT(NEWPOS);
      GOTOXY(X, Y);
      WRITE(LOOKSLIKE[FORWHAT]);
      SCREEN[X, Y] := FORWHAT;
   END;
END;

(*
 * PLACENEARBY FINDS A FREE COORDINATE ADJACENT TO
 * THE ARGUMENT COORDINATE AND PLACES THE THING THERE
 *)

PROCEDURE PLACENEARBY(VAR NEAR, COORD : COORDINATE);
VAR
   DELTAX, DELTAY : INTEGER;

BEGIN
   REPEAT
      REPEAT         (* FIND A RANDOM OFFSET FROM CURRENT COORDINATE *)
         DELTAX := RAND(-1, 1);
         DELTAY := RAND(-1, 1);
      UNTIL (DELTAX <> 0) OR (DELTAY <> 0);
      NEAR.X := COORD.X + DELTAX;
      NEAR.Y := COORD.Y + DELTAY;
   UNTIL (FREESPOT(NEAR) OR  (* MAKE SURE THAT IT DOESN'T *)
          (NEAR = PLAYER));  (* OVERRUN THE MONEY OR EXIT *)
   GOTOXY(NEAR.X, NEAR.Y);
   SCREEN[NEAR.X][NEAR.Y] := SNAKETHING;
   WRITE(LOOKSLIKE[SNAKETHING]);
END; (* PLACENEARBY *)
```

```
(*
 * REMOVES WHATEVER IS AT THE COORDINATES FROM
 * THE TERMINAL SCREEN AND THE ARRAY SCREEN.
 *)
PROCEDURE REMOVE(POS : COORDINATE);
BEGIN
   GOTOXY(POS.X, POS.Y);
   WRITE(' ');                  (* ERASE OLD MARK *)
   SCREEN[POS.X, POS.Y] := EMPTYTHING;
END;

PROCEDURE TAKEGOLD;
BEGIN
   SCORE := SCORE + MONEYWORTH;
   GOTOXY(0, 0);
   WRITE('$', SCORE);
   SCREEN[MONEY.X, MONEY.Y] := EMPTYTHING;
   MAKESPACE(MONEY, MONEYTHING);
END;

(*
 * POSITION ALL OF THE ITEMS IN THE GAME
 * MAKING SURE THAT NONE OF THEM OVERLAP
 *)

PROCEDURE PLACEOBJECTS;
VAR
   SNAKEBODY : INTEGER;
BEGIN
   MAKESPACE(SNAKE[1], SNAKETHING);
   FOR SNAKEBODY := 2 TO SNAKELENGTH DO
      PLACENEARBY(SNAKE[SNAKEBODY], SNAKE[SNAKEBODY - 1]);
   MAKESPACE(PLAYER, PLAYERTHING);
   MAKESPACE(MONEY, MONEYTHING);
   MAKESPACE(DOOR, DOORTHING);
END;

(*
 * READ THE PLAYER'S MOVE FROM KEYBOARD, NOT INPUT SO
 * THAT THE LETTER WILL NOT BE ECHOED AND MESS UP
 * THE DISPLAY
 *)

PROCEDURE PLAYERMOVE;
VAR
   COMMAND : CHAR;
   OLDPOS  : COORDINATE;
BEGIN
   OLDPOS := PLAYER;
   READ(KEYBOARD, COMMAND);
   WITH PLAYER DO
   BEGIN
      CASE COMMAND OF
         UPCOMMAND:
            IF Y > 0 THEN
               Y := Y - 1;
         DOWNCOMMAND:
            IF Y < HEIGHT THEN
               Y := Y + 1;
         LEFTCOMMAND:
            IF X > 0 THEN
               X := X - 1;
         RIGHTCOMMAND:
            IF X < WIDTH THEN
               X := X + 1;
      END; (* CASE *)
      IF SCREEN[X, Y] = SCORETHING THEN (* DON'T ALLOW HIM TO GO *)
         PLAYER := OLDPOS               (* ON TOP OF THE SCORE   *)
      ELSE
      BEGIN
         REMOVE(OLDPOS);           (* ERASE OLD POSITION *)
         IF PLAYER = MONEY THEN
            TAKEGOLD
         ELSE IF PLAYER = DOOR THEN
            LEFT := TRUE;
         GOTOXY(X, Y);             (* PLOT PLAYER'S NEW POSITION *)
         WRITE(PLAYERCHAR);
         SCREEN[X, Y] := PLAYERTHING;
      END;
   END; (* WITH PLAYER *)
END;

(*
 * USED BY SNAKEMOVE TO FIGURE OUT WHICH
 * WAY IS THE DIRECTION TOWARD THE PLAYER
 *)
FUNCTION SIGN(X : INTEGER) : INTEGER;
BEGIN
   IF X = 0 THEN
      SIGN := 0
   ELSE IF X > 0 THEN
      SIGN := 1
   ELSE
      SIGN := -1;
END;

(*
 * SNAKE MOVES RANDOMLY AT FIRST, THEN IT
 * GOES MORE DIRECTLY TOWARD THE PLAYER
 *)

PROCEDURE SNAKEMOVE;
VAR
   NEWPOS : COORDINATE;
   BODYPART : INTEGER;
```

```
BEGIN
   IF RAND(0, SCORE) <= 100 THEN
   PLACENEARBY(NEWPOS, SNAKE[1])
   ELSE
   BEGIN
      NEWPOS.X := SNAKE[1].X + SIGN(PLAYER.X - SNAKE[1].X);
      NEWPOS.Y := SNAKE[1].Y + SIGN(PLAYER.Y - SNAKE[1].Y);
      IF (SCREEN[NEWPOS.X, NEWPOS.Y] = EMPTYTHING) OR   (* DON'T OVERWRITE *)
         (NEWPOS = PLAYER) THEN                         (* DOOR OR MONEY   *)
      BEGIN
         GOTOXY(NEWPOS.X, NEWPOS.Y);
         WRITE(SNAKECHAR);
         SCREEN[NEWPOS.X, NEWPOS.Y] := SNAKETHING;
      END
      ELSE                              (* COULDN'T GO DIRECTLY TOWARD   *)
         PLACENEARBY(NEWPOS, SNAKE[1]); (* PLAYER, SO JUST MOVE RANDOMLY *)
   END;
   REMOVE(SNAKE[SNAKELENGTH]);          (* MOVE THE SNAKE BY ERASING THE *)
   IF NEWPOS = PLAYER THEN              (* TAIL AND MAKING A NEW HEAD    *)
      EATEN := TRUE;
   FOR BODYPART := SNAKELENGTH DOWNTO 2 DO   (* UPDATE COORDINATES OF    *)
   BEGIN                                     (* THE INSIDES OF THE SNAKE *)
      SNAKE[BODYPART] := SNAKE[BODYPART - 1];
      IF SNAKE[BODYPART] = PLAYER THEN  (* PLAYER MIGHT RUN *)
         EATEN := TRUE;                 (* INTO THE SNAKE   *)
   END;
   SNAKE[1] := NEWPOS;
END;


BEGIN           (* MAIN PROGRAM OF SNAKE *)
   INITIALIZE;
   PLACEOBJECTS;
   REPEAT
      PLAYERMOVE;
      IF NOT LEFT THEN
         SNAKEMOVE;
   UNTIL LEFT OR EATEN;
   GOTOXY(0, HEIGHT);           (* BOTTOM LEFT CORNER *)
   WRITELN;
   IF LEFT THEN
      WRITELN('YOU HAVE ESCAPED WITH $', SCORE)
   ELSE
      WRITELN('THE SNAKE HAS EATEN YOU');
END.
```

# HANGMAN

**The Game.** Hangman is, of course, that great old standby of a rainy-day game where you try to guess a word before your opponent can draw a stick figure being hanged at the gallows. In this version, the computer chooses the word (from a special "dictionary" file, which we will discuss below) and you try to guess it in time.

At the beginning of the game, the computer displays a line of hyphens representing the letters of the word that it has chosen. When you see the prompt:

```
NEXT GUESS?
```

you begin guessing letters, one at a time. If a letter is correct (i.e., if it is in the word the computer has chosen) then the appropriate hyphen (or hyphens) will be replaced by the letter. If you guess a letter that is not in the word, the computer swiftly draws part of the stick figure at the right of the screen—first the head, then the shoulders, the right arm, the left arm, the torso, the right leg, and finally the left leg. You are thus allowed seven incorrect guesses before the stick figure is completed and you lose the game. To make the guessing a little easier, the computer also displays the list of the letters that you have already guessed. If you complete the word before the man is hanged, then you win the game. The computer, programmed to be a sore loser, expresses itself appropriately:

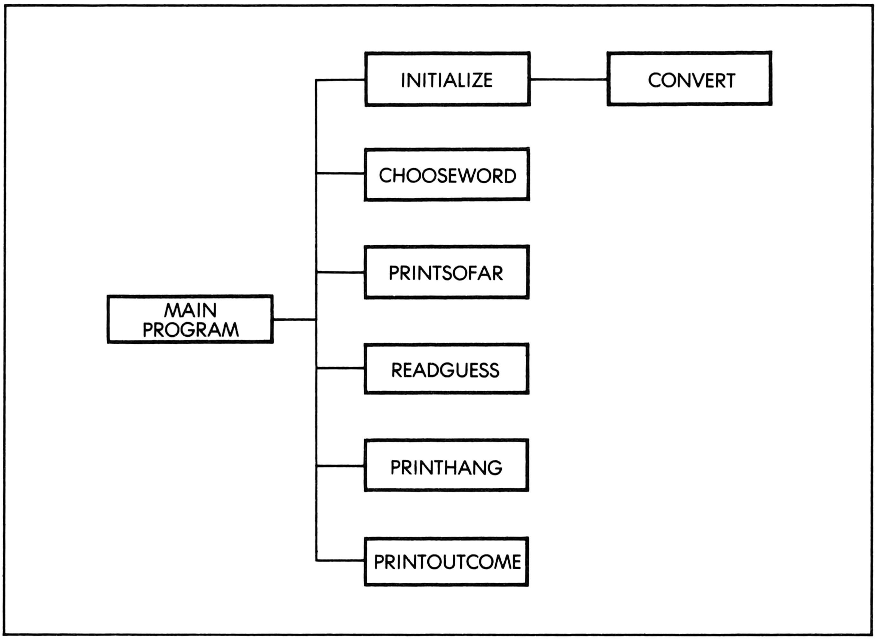
```
RATS!! YOU GUESSED IT!
```

Before you play the Hangman game, make sure you have run its companion program, MAKEDICT. MAKEDICT creates a file of words, under the file name DICTIONARY. Program HANGMAN uses this file to choose a word for the Hangman game.

Run MAKEDICT and read the instructions. Enter as many words as you want, in any order. (HANGMAN chooses a word at random out of the file.) The words cannot be more than 10 letters long.

```
           C O - P U T - R        0
                                 ---
           ABCINOPRTUX          / I \
                               /  I  \

     NEXT  GUESS?
```

# The Program.



MAIN PROGRAM
INITIALIZE
CONVERT
CHOOSEWORD
PRINTSOFAR
READGUESS
PRINTHANG
PRINTOUTCOME

In this program, two sets are used to test every letter that the player inputs:

```
GUESSES, WORDSLETTERS: SET OF 'A'..'Z';
```

These two sets are both initialized to the empty set in procedure INITIALIZE. Then, in procedure CHOOSEWORD, each letter of the word (AWORD) that the computer has chosen (at random) from DICTIONARY is added in turn to the set WORDSLETTERS:

```
FOR I := 1 TO LENGTH(AWORD) DO
  IF NOT (AWORD[I] IN WORDSLETTERS) THEN
    WORDSLETTERS := WORDSLETTERS + [AWORD[I]];
```

Procedure READGUESS prompts the player to input a guess and then reads the guess into the variable THISGUESS. If THISGUESS already belongs to the set GUESSES:

```
IF THISGUESS IN GUESSES
```

then the player has already tried the letter; an appropriate error message is displayed. Another error message is displayed if the guess is any character other than a letter:

```
IF NOT (THISGUESS IN ['A'..'Z'])
```

If the guess passes both of these tests, however, it is added to the set GUESSES:

```
GUESSES := GUESSES + [THISGUESS];
```

The final test is to determine if the guess is a good one (i.e., if the letter is in the word). If it is not, then TURNS (the number of guesses the player has already made) is incremented:

```
IF NOT (THISGUESS IN WORDSLETTERS) THEN
  TURNS := TURNS + 1;
```

Procedure PRINTSOFAR tests every letter of AWORD for membership in the set GUESSES. If a letter of the word has been guessed it is written, and CORRECTCOUNT is incremented:

```
IF AWORD[WORDINDEX] IN GUESSES THEN
   BEGIN
      WRITE (AWORD[WORDINDEX]);
      CORRECTCOUNT := CORRECTCOUNT + 1;
   END
```

Otherwise, a hyphen is written in the letter's place:

```
ELSE
   WRITE('-');
```

(Notice that PRINTSOFAR is called once before any guesses have been made to print out the correct number of hyphens representing the letters of the word.)

PRINTSOFAR also displays the letters that have been guessed. Since the elements of a set cannot be referenced individually, the set must be written in a roundabout way:

```
FOR GUESSINDEX := 'A' TO 'Z' DO
   IF GUESSINDEX IN GUESSES THEN
      WRITE(GUESSINDEX);
```

Finally, PRINTSOFAR checks to see if CORRECTCOUNT (the number of correct guesses) is equal to the length of the word. If it is, then the BOOLEAN variable GUESSEDIT is set to TRUE.

PRINTHANG displays the hangman. A **CASE** statement, with the **CASE** "selector" TURNS, is used to display each part of the man at the appropriate moment. (Thus, when TURNS = 7, the man will be complete.) GOTOXY is used to position the cursor properly before each WRITE statement.

The main program continues calling READGUESS, PRINTSOFAR, and PRINTHANG until either the word has been guessed (GUESSEDIT) or the turns have been used up (TURNS > MAXTURNS). Procedure PRINTOUTCOME is called to display a farewell message.

Program MAKEDICT and procedures INITIALIZE and CHOOSEWORD (of program HANGMAN) all use the UCSD Pascal file-handling procedures. The task of MAKEDICT is to set up an external file for the words that HANGMAN will use. Thus, DICT is defined in the declaration section:

```
TYPE
   WORD = STRING[STRINGSIZE];
VAR
   DICT: FILE OF WORD;
```

The procedure REWRITE sets the *file marker* to the beginning of the file (abandoning any information already in the file):

```
REWRITE(DICT,'DICTIONARY');
```

Notice that the UCSD version of REWRITE allows two parameters: a file variable and a string. (The string specifies the name of the external file.)

We want the first record of the DICTIONARY file to contain the size of the file. (We will see why this information is needed when we return to procedure CHOOSEWORD.) However, since DICT is a file of WORD, the integer indicating the size must be converted into a string before it can be stored as a record of the file. The UCSD function STR performs this task. This record counter is initialized to 0, converted into a string, and stored in the variable NEXTWORD:

```
STR(0,NEXTWORD);
```

Next, the current record of DICT is assigned the value NEXTWORD:

```
DICT^ := NEXTWORD;
PUT(DICT);
```

PUT also moves the marker to the next position of the file. As we will see, the positions of a file are numbered sequentially, beginning with 0. (Thus, the zeroth position of this file will contain a record indicating the length of the file.)

The creation of the rest of the file is carried out in a similar way. The player enters a word, which is read into the variable NEXTWORD:

```
READLN(NEXTWORD);
```

The value of NEXTWORD is assigned to the buffer variable DICT^, and the record is written to the file:

```
DICT^ := NEXTWORD;
PUT(DICT);
```

For each record that is written, NUMWORDS is incremented by 1:

```
NUMWORDS := NUMWORDS + 1;
```

Finally, when the process of writing into the file is complete (i.e., EOF has been entered) the UCSD SEEK procedure is used to move the marker back to the zeroth position of the file:

```
SEEK(DICT,0);
```

NUMWORDS is converted into a string and assigned to NEXTWORD:

```
STR(NUMWORDS,NEXTWORD);
```

and the value becomes the zeroth record of the file:

```
DICT^ := NEXTWORD;
PUT(DICT);
```

The last statement of the program closes the file. (The UCSD LOCK option makes the file permanent.):

```
CLOSE(DICT,LOCK);
```

Now let us return briefly to program HANGMAN to see how the file is read.

In procedure INITIALIZE, a call to the UCSD procedure RESET opens the file, sets the marker to the beginning of the file, and assigns the record of that position to DICT^:

```
RESET(DICT,'DICTIONARY');
```

The value of DICT^ is next assigned to AWORD, and procedure CONVERT changes AWORD back into an integer (the number of records in the file):

```
AWORD := DICT^;
NUMWORDS := CONVERT(AWORD);
```

In procedure CHOOSEWORD, a random number between 1 and NUMWORDS is chosen:

```
NTHWORD := 1 + RANDOM MOD NUMWORDS;
```

The NTHWORD record of the file is accessed directly via the UCSD SEEK procedure. SEEK positions the marker of file DICT so that the following GET will access the NTHWORD record:

```
SEEK(DICT,NTHWORD);
GET(DICT);
```

Finally, the record there is assigned to AWORD:

```
AWORD := DICT^;
```

This value of AWORD is, of course, the word that the player will be trying to guess in the Hangman game.

```
PROGRAM HANGMAN;

(*
 * THIS PROGRAM PLAYS THE GAME OF HANGMAN
 *)

USES APPLESTUFF;            (* FOR RANDOM NUMBERS *)

CONST
   STRINGSIZE = 10;
   MAXTURNS   = 7;          (* NUMBER OF TURNS TO GUESS WORD *)
   BACKSLASH  = 220;        (* ASCII CODE FOR BACKWARD SLANT *)

TYPE
   WORD = STRING [STRINGSIZE];

VAR
   DICT : FILE OF WORD;
   AWORD : WORD;
   GUESSES, WORDSLETTERS : SET OF 'A' .. 'Z';
   NUMWORDS : INTEGER;
   TURNS : INTEGER;
   GUESSEDIT : BOOLEAN;

(*
 * CONVERT : CONVERTS A STRING
 * TO THE NUMBER THAT IT REPRESENTS.  ALL
 * NON-NUMERIC CHARACTERS ARE IGNORED.
 *)
```

```
FUNCTION CONVERT(NUMBER : WORD) : INTEGER;
VAR
   RESULT : INTEGER;
   STRINDEX : INTEGER;

BEGIN
   RESULT := 0;
   FOR STRINDEX := 1 TO LENGTH(AWORD) DO
      IF NUMBER[STRINDEX] IN ['0' .. '9'] THEN
         RESULT := RESULT * 10 + ORD(NUMBER[STRINDEX]) - ORD('0');
   CONVERT := RESULT;
END; (* CONVERT *)

PROCEDURE INITIALIZE;
BEGIN
   RESET(DICT, 'DICTIONARY');
   AWORD := DICT^;
   NUMWORDS := CONVERT(AWORD);
   RANDOMIZE;
   GUESSES := [];
   WORDSLETTERS := [];
   TURNS := 0;
   GUESSEDIT := FALSE;
END; (* INITIALIZE *)


(*
 * THIS PROCEDURE RANDOMLY PICKS A WORD OUT OF
 * THE HANGMAN DICTIONARY
 *)


PROCEDURE CHOOSEWORD;
VAR
   NTHWORD,I : INTEGER;

BEGIN
   NTHWORD := 1 + RANDOM MOD NUMWORDS;
   SEEK(DICT, NTHWORD);
   GET(DICT);
   AWORD := DICT^;
   FOR I := 1 TO LENGTH(AWORD) DO
      IF NOT (AWORD[I] IN WORDSLETTERS) THEN
         WORDSLETTERS := WORDSLETTERS + [AWORD[I]];
END; (* CHOOSEWORD *)

(*
 * READGUESS : ASKS FOR ANOTHER LETTER THAT
 * THE PLAYER EXPECTS TO BE IN THE WORD.
 * A LETTER THAT ALREADY HAS BEEN GUESSED
 * IS NOT ACCEPTED
 *)
```

```
PROCEDURE READGUESS;
VAR
   THISGUESS : CHAR;
   INVALID, ERASE : BOOLEAN;
BEGIN
   INVALID := TRUE;
   ERASE := FALSE;
   WHILE INVALID DO
   BEGIN
      GOTOXY(1,10);
      WRITE('NEXT GUESS ? ');
      READLN(THISGUESS);
      IF EOF THEN
         EXIT(HANGMAN)
      ELSE IF THISGUESS IN GUESSES THEN         (* ALREADY TRIED IT *)
      BEGIN
         GOTOXY(1,12);
         WRITE('YOU HAVE ALREADY USED ''', THISGUESS, '''.');
         ERASE := TRUE;
      END
      ELSE IF NOT(THISGUESS IN ['A'..'Z']) THEN (* COMPLAIN IF NOT *)
      BEGIN                                     (* ALPHABETIC       *)
         GOTOXY(1,12);
         WRITE('ONLY LETTERS ARE ALLOWED.');
         ERASE := TRUE;
      END
      ELSE                                      (* A VALID GUESS    *)
      BEGIN
         GUESSES := GUESSES + [THISGUESS];
         INVALID := FALSE;
      END;
   END; (* WHILE *)
   IF ERASE THEN
   BEGIN
      GOTOXY(1,12);
      WRITE(' ':27);
   END;
   IF NOT (THISGUESS IN WORDSLETTERS) THEN
      TURNS := TURNS + 1;
END; (* READGUESS *)

(*
 * PRINTSOFAR : PROCEDURE THAT PRINTS OUT THE
 * WORD BEING GUESSED.  LETTERS THAT AREN'T
 * KNOWN YET ARE PRINTED AS '-'.  THE LETTERS
 * THAT HAVE BEEN GUESSED ARE PRINTED UNDERNEATH
 *)
```

```
PROCEDURE PRINTSOFAR;
VAR
   WORDINDEX : INTEGER;
   GUESSINDEX : CHAR;
   CORRECTCOUNT : INTEGER;

BEGIN
   CORRECTCOUNT := 0;
   GOTOXY(10,5);
   FOR WORDINDEX := 1 TO LENGTH(AWORD) DO
   BEGIN
      IF AWORD[WORDINDEX] IN GUESSES THEN
      BEGIN
         WRITE(AWORD[WORDINDEX]);
         CORRECTCOUNT := CORRECTCOUNT + 1;
      END
      ELSE
         WRITE('-');
      WRITE(' ');
   END;
   GOTOXY(10,7);
   FOR GUESSINDEX := 'A' TO 'Z' DO
      IF GUESSINDEX IN GUESSES THEN
         WRITE(GUESSINDEX);
   IF CORRECTCOUNT = LENGTH(AWORD) THEN
      GUESSEDIT := TRUE;
END; (* PRINTSOFAR *)

(*
 * PRINTHANG : PRINTS THE HANGMAN
 *)

PROCEDURE PRINTHANG;

BEGIN
   CASE TURNS OF
      1 :BEGIN
            GOTOXY(30,5);
            WRITE('O');
         END;
      2 :BEGIN
            GOTOXY(29,6);
            WRITE('---');
         END;
      3 :BEGIN
            GOTOXY(28,7);
            WRITE('/');
            GOTOXY(27,8);
            WRITE('/');
         END;
      4 :BEGIN
            GOTOXY(32,7);
            WRITE(CHR(BACKSLASH));
            GOTOXY(33,8);
            WRITE(CHR(BACKSLASH));
         END;
      5 :BEGIN
            GOTOXY(30,7);
            WRITE('I');
            GOTOXY(30,8);
            WRITE('I');
         END;
      6 :BEGIN
            GOTOXY(29,9);
            WRITE('/');
            GOTOXY(28,10);
            WRITE('/');
         END;
      7 :BEGIN
            GOTOXY(31,9);
            WRITE(CHR(BACKSLASH));
            GOTOXY(32,10);
            WRITE(CHR(BACKSLASH));
         END;
   END; (* CASE *)
END; (* PRINTHANG *)

PROCEDURE PRINTOUTCOME;
BEGIN
   GOTOXY(5,16);
   IF GUESSEDIT THEN
   BEGIN
      WRITELN('RATS!! YOU GUESSED IT!');
   END
   ELSE
   BEGIN
      WRITELN('THE WORD WAS :',AWORD);
      WRITELN;
      WRITELN('DUE TO THE VIOLENT NATURE OF');
      WRITELN('YOUR DEMISE THE REST OF THE SCENE');
      WRITELN('HAS BEEN CENSORED.');
   END
END; (* PRINTOUTCOME *)

BEGIN (* MAIN PROGRAM *)
   INITIALIZE;
   CHOOSEWORD;
   PRINTSOFAR;
   WHILE NOT GUESSEDIT AND (TURNS < MAXTURNS) DO
   BEGIN
      READGUESS;
      PRINTSOFAR;
      PRINTHANG;
   END;
   PRINTOUTCOME;
END. (* HANGMAN *)
```

```
PROGRAM MAKEDICT;

(*
 * THIS PROGRAM MAKES THE DATAFILE
 * THAT IS USED BY THE HANGMAN PROGRAM.
 *)

CONST
   STRINGSIZE = 10;     (* MAX LENGTH OF WORDS USABLE *)

TYPE
   WORD = STRING[STRINGSIZE];

VAR
   DICT : FILE OF WORD;
   NEXTWORD : WORD;
   NUMWORDS : INTEGER;

(*
 * THE DICTIONARY FOR HANGMAN IS A FILE OF STRINGS.
 * IT CANNOT BE A TEXT FILE BECAUSE SEEKING IS NOT
 * ALLOWED ON TEXT, AND HANGMAN USES A RANDOM SEEK
 * TO GET THE WORD.  THE FIRST STRING IN THE FILE
 * CONTAINS THE NUMBER OF WORDS IN THE DICTIONARY.
 *)

BEGIN
   NUMWORDS := 0;
   REWRITE(DICT, 'DICTIONARY');
   STR(0, NEXTWORD);
   DICT^ := NEXTWORD;           (* ALLOCATE A RECORD IN WHICH TO  *)
   PUT(DICT);                   (* PUT THE SIZE OF THE DICTIONARY *)
   WRITELN('TYPE THE WORDS TO BE ENTERED INTO THE');
   WRITELN('DICTIONARY; NO MORE THAN ', STRINGSIZE:1, ' LETTERS LONG.');
   WRITELN('TYPE THE END-OF-FILE CHARACTER TO EXIT.');
   READLN(NEXTWORD);
   WHILE NOT EOF AND (LENGTH(NEXTWORD) > 0) DO
   BEGIN
      DICT^ := NEXTWORD;
      PUT(DICT);
      NUMWORDS := NUMWORDS + 1;
      READLN(NEXTWORD);
   END;
   WRITELN('THAT''S ALL, FOLKS!');
   SEEK(DICT, 0);
   STR(NUMWORDS, NEXTWORD);
   DICT^ := NEXTWORD;
   PUT(DICT);
   CLOSE(DICT, LOCK);
END.
```

# LIFE

**The Game.** The Game of Life, now a classic among computer games, examines the patterns by which populations wax and wane. The rules that govern these patterns are supplied by the game itself. The game traces the life history of a colony of "cells," generation by generation. During each generation, the birth, life, and death of each individual of the colony depend completely upon the number of cells that are living in the immediate environment of that individual.

Your input to this program occurs at the very beginning of the game, to create the first generation of the colony. You will be prompted:

```
POSITION OF CELL #1 IS:
```

You must answer this prompt with two integers: the coordinates of the position where you wish to create a live cell. (Put a space, *not a comma*, between the coordinates.) Each cell goes on the screen, which is 24 positions high by 40 positions wide. (You should enter integral coordinates from 0 to 23 and from 0 to 39.) If you enter a set of coordinates, and then change your mind, enter the same coordinates a second time to "erase" the cell at that position. The first generation of the colony may contain as many or as few individuals as you wish. When you have completed the input of cells, enter any set of coordinates that falls outside the 24-by-40 screen (for example, 30 50) to signal that input is finished.

By the nature of the square grid pattern that the colony lives in, each individual can have a maximum of eight neighbors in its immediate surroundings—neighbors above it and below it, to its right and to its left, and four neighbors in the diagonal positions. In this game, the life of an individual depends on whether or not any of these eight positions actually contain live cells. For the creation of each new generation, then, the rules of Life are as follows:

- If a position contains a live cell, that cell will survive into the next generation if it has exactly two or three neighbors. Otherwise, in the words of this program, it dies either of loneliness or of overcrowding.
- If a position contains no living cell, a cell will be born in that position in the next generation if the position has three living cell neighbors.

Two phenomena commonly occur in colonies governed by these rules. A colony can die out altogether, or a colony can stabilize, so that births and deaths no longer occur. If either one of these events takes place, the Game of Life ends. A third phenomenon may also occur. A colony can enter into a repeating cycle of generation patterns. If this should happen, or if for any other reason you wish to stop the program, press any key on the keyboard during the generation process, and the game will come to an end.

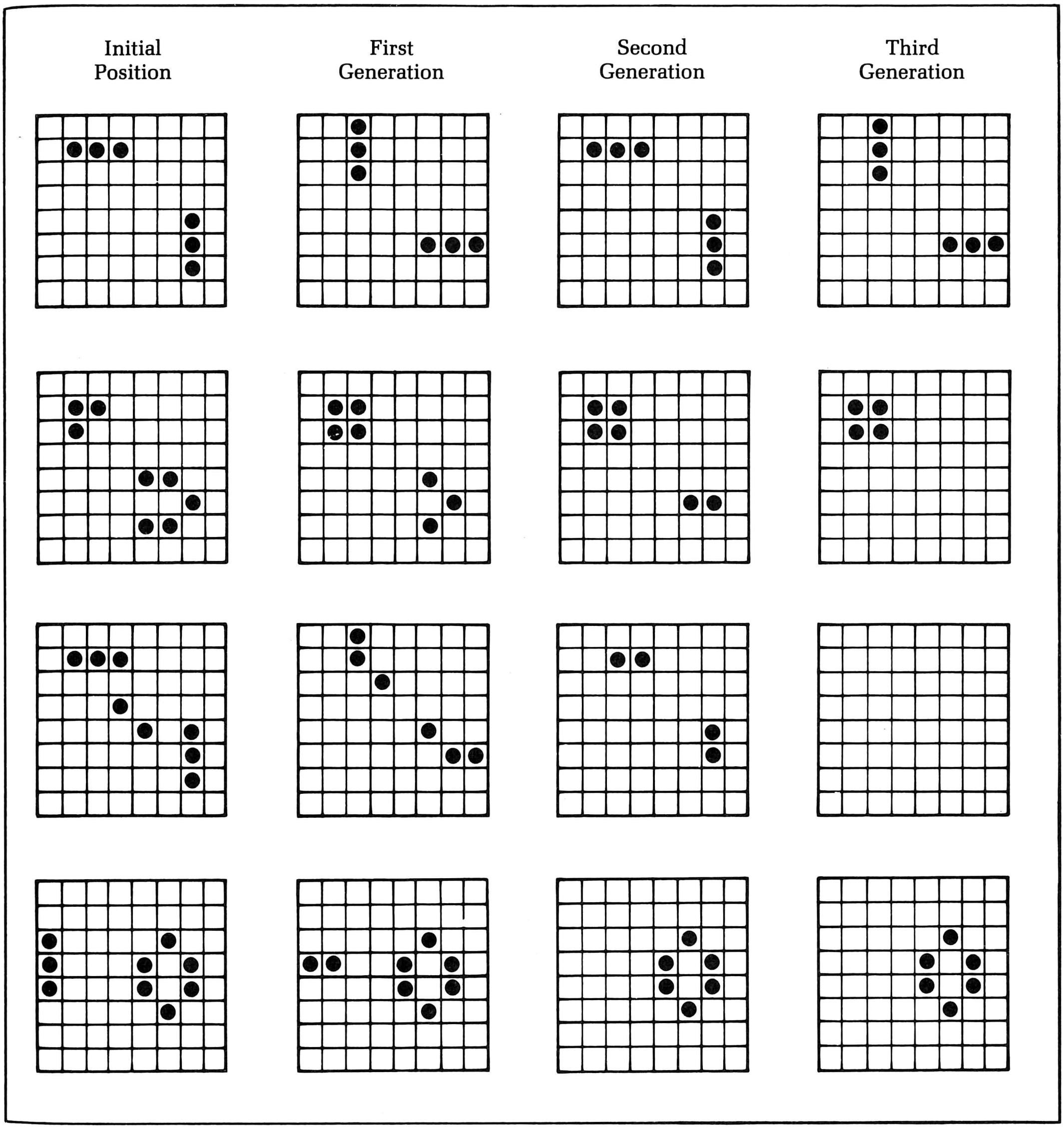
Experimentation with this game will reveal the range of fascinating growth patterns that result from different first-generation colonies.

# The Program.

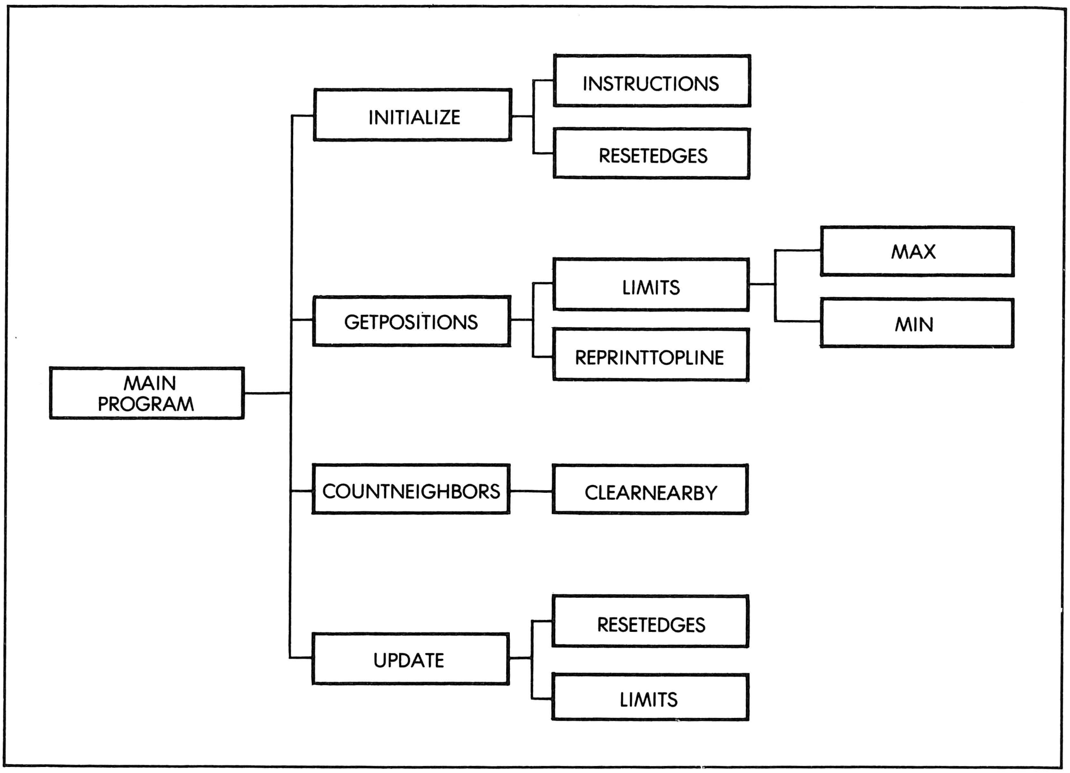
The universe in which the population will live and die is represented in the program by the two-dimensional array BOARD. BOARD is an array of CELL; CELL is a record type with two fields:

```
CELL = RECORD
  LOOKSLIKEITIS: STATE;
  NEARBY: INTEGER;
END;
```

The field LOOKSLIKEITIS, of type STATE, indicates whether or not a given element of BOARD is alive:


Initial Position
First Generation
Second Generation
Third Generation

***Game of Life examples. From* Inside Basic Games *by Richard Mateosian (SYBEX, 1981).***


MAIN PROGRAM
INITIALIZE
INSTRUCTIONS
RESETEDGES
GETPOSITIONS
LIMITS
MAX
MIN
REPRINTTOPLINE
COUNTNEIGHBORS
CLEARNEARBY
UPDATE
RESETEDGES
LIMITS

```
STATE = (ALIVE, DEAD);
```

The field NEARBY is an integer that records the number of living neighbors a cell has.

The other important record type is EDGES:

```
EDGES = RECORD
   LEFT,RIGHT,
   TOP,BOTTOM: INTEGER;
END;
```

The variable EDGE is of type EDGES. As we will see, this variable keeps track of the boundaries of the rectangular area occupied by the population at any given time. A good deal of time can thus be saved between the production of each generation; rather than investigate all the elements of BOARD, the program need concern itself only with the elements within the rectangle delineated by the values of EDGE. (We will also see that EDGE is updated with each new generation.)

Before the generation process can begin, the main program sets up the game via calls to INITIALIZE and GETPOSITIONS.

Procedure INITIALIZE calls procedure INSTRUCTIONS to display a description of the game. Then, within two levels of **FOR** loops, all the elements of BOARD are initialized:

```
BOARD[DOWN,ACROSS].LOOKSLIKEITIS := DEAD;
BOARD[DOWN,ACROSS].NEARBY := 0;
```

In other words, there are no living cells, and, of course, no living neighbors. Similarly, a call to procedure RESETEDGES initializes the fields of the record variable EDGE:

```
EDGE.TOP := HEIGHT - 1;
EDGE.RIGHT := MINBOUND + 1;
EDGE.LEFT := WIDTH - 1;
EDGE.BOTTOM := MINBOUND + 1;
```

At first glance these initializations may seem unusual. If the origin (0,0) is located at the upper left corner of the screen, why are we setting the right and bottom edges to 0 and the left and top edges to the maximum width and height, respectively? We can quickly answer this question by jumping ahead to procedure LIMITS, which is called each time a new cell is born:

```
WITH EDGE DO
  BEGIN
    LEFT := MIN(LEFT,X);
    RIGHT := MAX(RIGHT,X);
    TOP := MIN(TOP,Y);
    BOTTOM := MAX(BOTTOM,Y);
  END;
```

(The functions MAX and MIN simply return the larger and smaller of the two values they are passed.) We can see that, due to the initializations in RESETEDGES, the coordinates of the first living cell that is entered onto the screen are guaranteed to change the values of all four fields of EDGE. Each cell that is born thereafter will change one or more of the fields only if it sets a new outer boundary for the rectangle the colony lives in.

Procedure GETPOSITIONS next takes control to accept the player's input for the first-generation colony. A GOTOXY statement places the cursor at the top right corner of the screen, where a prompt is written. A **WHILE** loop controls the input process. Each set of coordinates that is input is first tested to see if the point lies within the boundaries of the screen. If it does not, it is taken as a signal that all the points of the first generation have been entered; the BOOLEAN variable FINISHED is set to TRUE, and the **WHILE** loop ends.

If the point is within the boundaries of the screen, however, a call to LIMITS resets the field values of EDGE, a GOTOXY moves the cursor to the appropriate position on the screen, and the field LOOKSLIKEITIS is tested to see if the cell at that position is ALIVE or DEAD. If it is already ALIVE, it is reset to DEAD, the cell is "erased" from the screen (by writing a blank character) and POPULATION is decremented:

```
IF LOOKSLIKEITIS = ALIVE THEN
BEGIN
   WRITE(' ');
   LOOKSLIKEITIS := DEAD;
   POPULATION := POPULATION - 1;
END
```

Otherwise, the reverse process takes place. The cell (represented by an asterisk) is written, LOOKSLIKEITIS is reset, and POPULATION is incremented:

```
ELSE
BEGIN
   WRITE('*');
   LOOKSLIKEITIS := ALIVE;
   POPULATION := POPULATION + 1;
END;
```

Then the cursor is moved back to the origin, and a prompt is written for input of the next cell. At the end of procedure GETPOSITIONS, a call to procedure REPRINTTOPLINE displays any cells that may have been hidden by the prompt on the first line.

Next in the main program, a **REPEAT/UNTIL** loop makes continued alternate calls to COUNTNEIGHBORS and UPDATE until the population has completely died out:

```
(POPULATION = 0)
```

or the population has stabilized:

```
((BIRTHS = 0) AND (DEATHS = 0))
```

or, finally, until the player stops the program by pressing a key on the keyboard:

```
KEYPRESS
```

(If one of the former two events takes place, an explanatory message is displayed, and the program terminates.)

As its name implies, procedure COUNTNEIGHBORS counts up the number of living neighboring cells for each live member of the colony, and assigns that value to the appropriate NEARBY field of BOARD. Since every cell within the boundaries set by EDGE will be investigated, and any or all of the NEARBY fields might change, a call to CLEARNEARBY sets all of these fields to 0:

```
BOARD[DOWN,ACROSS].NEARBY := 0;
```

COUNTNEIGHBORS then performs its task within four levels of **FOR** loops. The two outer loops increment the indices of BOARD; if a cell is alive, the two inner loops increment the NEARBY field of each cell within the 3-by-3 matrix of positions that surround the original cell:

```
IF BOARD[DOWN,ACROSS].LOOKSLIKEITIS = ALIVE THEN
  FOR DELTADOWN := -1 TO 1 DO
    FOR DELTAACROSS := -1 TO 1 DO
      BOARD[DOWN + DELTADOWN,
              ACROSS + DELTAACROSS].NEARBY :=
      BOARD[DOWN + DELTADOWN,
              ACROSS + DELTAACROSS].NEARBY + 1;
```

As a comment to this procedure explains, the middle cell of the 3-by-3 matrix (i.e., the very cell that is being investigated) is counted, incorrectly, as a neighbor of itself. However, we will see that this "error" (which considerably simplifies the code) is corrected in procedure UPDATE.

UPDATE uses two nested **FOR** loops and a **CASE** statement to display the new generation of the colony. As in procedure COUNTNEIGHBORS, the **FOR** loops increment the indices of BOARD; however, in UPDATE the functions MIN and MAX are used to avoid moving outside the limits of the screen. (In other words, cells that are born outside the screen are ignored.) LOOKSLIKEITIS is the "selector" of the **CASE** statement. If a cell is DEAD and has exactly three neighbors (NEARBY = 3) then the cell is born. If the cell is ALIVE and has two or three neighbors then it survives:

```
IF (NEARBY = 3) OR (NEARBY = 4) THEN . . .
```

Notice that this is where the discrepancy of COUNTNEIGHBORS is accounted for: NEARBY is tested for values of 3 and 4 rather than 2 and 3.

If the cell has too few or too many neighbors, then it dies a quick and silent death:

```
ELSE
BEGIN
  LOOKSLIKEITIS := DEAD;
  GOTOXY(ACROSS,DOWN);
  WRITE(' ');
  DEATHS := DEATHS + 1;
END;
```

```
PROGRAM LIFE;
USES
   APPLESTUFF;

CONST
   HEIGHT       = 24;         (* NUMBER OF LINES ON SCREEN   *)
   WIDTH        = 40;         (* NUMBER OF COLUMNS ON SCREEN *)
   MINBOUND     = -1;         (* MINIMUM DIMENSION OF SCREEN BOUNDS *)
   CLEARSCREEN  = 12;   (* PRINT CHR OF THIS CLEARS SCREEN *)

TYPE
   STATE = (ALIVE, DEAD);

   CELL = RECORD                (* EACH POSITION HAS A STATE  *)
             LOOKSLIKEITIS : STATE;
             NEARBY : INTEGER;  (* AND A COUNT OF THE NUMBER  *)
          END;                  (* OF ADJACENT LIVING CELLS   *)
   EDGES = RECORD               (* THE EDGES OF A COLONY ARE THE *)
             LEFT,              (* COORDINATES OF A SQUARE THAT  *)
             RIGHT,             (* WOULD CONTAIN THE PATTERN     *)
             TOP, BOTTOM : INTEGER;
          END;


VAR
   BOARD : ARRAY [MINBOUND .. HEIGHT] OF ARRAY [MINBOUND .. WIDTH] OF CELL;

   POPULATION, BIRTHS, DEATHS : INTEGER;
   CH : CHAR;           (* BUFFER FOR CHARACTER THAT USER MAY *)
                        (* TYPE TO 'INTERRUPT' THE CYCLE      *)
   EDGE : EDGES;

(*
 * INITIALIZES THE EDGES OF THE PATTERN. THIS STARTS REPRESENTING
 * A PATTERN WHICH HAS NO INSIDES; THE TOP IS LOWER THAN THE BOTTOM,
 * THE LEFT SIDE IS TO THE RIGHT OF THE RIGHT SIDE.   THIS ENSURES
 * THAT THE COORDINATES OF THE CORNER OF THE PATTERN AFTER IT
 * IS ENTERED WILL BE CORRECT WITHOUT NEEDING TO SCAN THE ENTIRE
 * ARRAY AFTER THE PATTERN IS ENTERED (A TIME-CONSUMING PROCESS).
 *)
PROCEDURE RESETEDGES;
BEGIN
   EDGE.TOP := HEIGHT - 1;
   EDGE.RIGHT := MINBOUND + 1;
   EDGE.LEFT := WIDTH - 1;
   EDGE.BOTTOM := MINBOUND + 1;
END;

PROCEDURE INSTRUCTIONS;
VAR
   ANSWER : CHAR;
BEGIN
   WRITELN('WOULD YOU LIKE INSTRUCTIONS FOR LIFE ?');
   WRITE('? ');
   READLN(ANSWER);
   WHILE NOT (ANSWER IN ['Y', 'N']) DO
   BEGIN
```

```
      WRITELN('PLEASE ANSWER YES OR NO.');
      WRITE('? ');
      READLN(ANSWER);
   END;
   IF ANSWER = 'Y' THEN
   BEGIN
      WRITELN('LIFE SIMULATES THE GROWTH OF A COLONY');
      WRITELN('OF ANIMALS ON A 0..', HEIGHT-1:1,
            ' BY 0..', WIDTH-1:1, ' ''WORLD''.');
      WRITELN('WHETHER A CELL IS BORN, LIVES, OR');
      WRITELN('DIES DEPENDS ON THE NUMBER OF LIVING');
      WRITELN('ANIMALS IMMEDIATELY ADJACENT TO IT.');
      WRITELN('IF A CELL IS EMPTY AND HAS EXACTLY 3');
      WRITELN('NEIGHBORS IT WILL BE BORN IN THE');
      WRITELN('NEXT GENERATION.  IF IT IS ALIVE AND');
      WRITELN('HAS EITHER TWO OR THREE NEIGHBORS IT');
      WRITELN('WILL CONTINUE TO LIVE.  OTHERWISE IT');
      WRITELN('DIES OF LONELINESS OR OVERCROWDING.');
      WRITELN('   THE INITIAL PATTERN IS ENTERED BY');
      WRITELN('TYPING THE ROW AND THEN THE COLUMN OF');
      WRITELN('THE DESIRED POSITION.  A CELL IS REMOVED');
      WRITELN('BY ENTERING ITS POSITION AGAIN. TO');
      WRITELN('FINISH ENTERING GIVE A POSITION ');
      WRITELN('OUTSIDE OF THE DIMENSIONS OF THE SCREEN');
      WRITELN('TO STOP A PATTERN JUST HIT ANY KEY.');
      WRITELN('TYPE RETURN TO START :');
      READLN;
   END;
   WRITE(CHR(CLEARSCREEN));
END;

(*
 * INITIALIZE : RESETS THE BOARD TO EMPTY ( ALL DEAD AND
 * WITH NO NEIGHBORS.)
 *)

PROCEDURE INITIALIZE;
VAR
   DOWN, ACROSS : INTEGER;
BEGIN
   INSTRUCTIONS;
   FOR DOWN := MINBOUND TO HEIGHT DO
      FOR ACROSS := MINBOUND TO WIDTH DO
      BEGIN
         BOARD[DOWN, ACROSS].LOOKSLIKEITIS := DEAD;
         BOARD[DOWN, ACROSS].NEARBY := 0;
      END;
   RESETEDGES;
END; (* INITIALIZE *)

(*
 * MAX (& MIN) : RETURNS THE LARGER (SMALLER) OF THE TWO
 * INTEGER ARGUMENTS
 *)
```

```
FUNCTION MAX(A, B : INTEGER) : INTEGER;
BEGIN
   IF A >= B THEN
      MAX := A
   ELSE
      MAX := B
END;

FUNCTION MIN(A, B : INTEGER) : INTEGER;
BEGIN
   IF A <= B THEN
      MIN := A
   ELSE
      MIN := B
END;

(*
 * DETERMINE IF AND HOW THE CO-ORDINATES PASSED AS ARGUMENT
 * CHANGE THE BOUNDS OF THE PATTERN ( THE POSITION OF A BOX THAT
 * COULD CONTAIN THE LIVING CELLS), CHECKING THAT IT DOES NOT
 * GO OFF ONE OF THE SIDES OF THE BOARD.
 *)
PROCEDURE LIMITS(X, Y : INTEGER);
BEGIN
   WITH EDGE DO
   BEGIN
      LEFT   := MIN(LEFT, X);
      RIGHT  := MAX(RIGHT, X);
      TOP    := MIN(TOP, Y);
      BOTTOM := MAX(BOTTOM, Y);
   END;
END;

(*
 * THIS ERASES THE RECORD OF THE NEIGHBORS
 * OF ALL THE CELLS, IN PREPARATION FOR THE
 * NEW CALCULATION OF THE NEARBY FIELD
 *)

PROCEDURE CLEARNEARBY;
VAR
   DOWN, ACROSS : INTEGER;
BEGIN
   FOR DOWN := EDGE.TOP - 1 TO EDGE.BOTTOM + 1 DO
      FOR ACROSS := EDGE.LEFT - 1 TO EDGE.RIGHT + 1 DO
         BOARD[DOWN][ACROSS].NEARBY := 0;
END; (* CLEARNEARBY *)

(*
 * COMPUTES THE NUMBER OF ADJACENT CELLS, AND THUS
 * WHICH CELLS WILL SURVIVE THROUGH THE NEXT GENERATION.
 * TO SPEED THIS UP, THE MIDDLE CELL OF THE 3 BY 3 MATRIX
 * WHICH IS BEING EXAMINED IS INCLUDED IN THE COUNT, EVEN THOUGH
 * IT IS NOT REALLY A NEIGHBOR OF ITSELF. THIS OFF-BY-ONE
 * DISCREPANCY IS TAKEN INTO ACCOUNT IN THE BOARD UPDATE.
 *)
```

```
PROCEDURE COUNTNEIGHBORS;
VAR
   DOWN, ACROSS : INTEGER;
   DELTADOWN, DELTAACROSS : INTEGER;
BEGIN
   CLEARNEARBY;         (* ERASE THE COUNT LEFT OVER FROM LAST GENERATION *)
   FOR DOWN := EDGE.TOP - 1 TO EDGE.BOTTOM + 1 DO
      FOR ACROSS := EDGE.LEFT - 1  TO EDGE.RIGHT + 1 DO
          IF BOARD[DOWN][ACROSS].LOOKSLIKEITIS = ALIVE THEN
             FOR DELTADOWN := -1 TO 1 DO
                FOR DELTAACROSS := -1 TO 1 DO
                   BOARD[DOWN + DELTADOWN][ACROSS + DELTAACROSS].NEARBY :=
                   BOARD[DOWN + DELTADOWN][ACROSS + DELTAACROSS].NEARBY + 1;
END; (* COUNTNEIGHBORS *)
(*
 * UPDATE
 * IF A BIRTH OR DEATH OCCURS THE SCREEN IS UPDATED.
 *)
PROCEDURE UPDATE;
VAR
   DOWN, ACROSS : INTEGER;
   LOCALEDGE : EDGES;
BEGIN
   BIRTHS := 0;
   DEATHS := 0;
   LOCALEDGE := EDGE;
   RESETEDGES;

   FOR DOWN := MAX(MINBOUND + 1, LOCALEDGE.TOP - 1) TO
               MIN(HEIGHT - 1, LOCALEDGE.BOTTOM  + 1) DO
      FOR ACROSS := MAX(MINBOUND + 1, LOCALEDGE.LEFT - 1) TO
                    MIN(WIDTH - 1, LOCALEDGE.RIGHT + 1) DO
         WITH BOARD[DOWN][ACROSS] DO
            CASE LOOKSLIKEITIS OF
            DEAD:
               IF NEARBY = 3 THEN
               BEGIN
                  LOOKSLIKEITIS := ALIVE;
                  GOTOXY(ACROSS, DOWN);
                  WRITE('*');
                  LIMITS(ACROSS, DOWN);
                  BIRTHS := BIRTHS + 1;
               END;
            ALIVE:
               IF (NEARBY = 3) OR (NEARBY = 4) THEN
                  LIMITS(ACROSS, DOWN)
               ELSE
               BEGIN
                  LOOKSLIKEITIS := DEAD;
                  GOTOXY(ACROSS, DOWN);
                  WRITE(' ');
                  DEATHS := DEATHS + 1;
               END;
            END; (* CASE *)
   POPULATION := POPULATION + BIRTHS - DEATHS;
END; (* UPDATE *)
```

```
(*
 * GET THE STARTING POSITIONS OF THE CELLS
 *)
PROCEDURE GETPOSITIONS;
VAR
   DOWN, ACROSS : INTEGER;
   FINISHED : BOOLEAN;

   (*
    * THIS IS NEEDED TO REPRINT THE TOP LINE OF THE PATTERN, WHICH
    * IS DESTROYED BY THE PROMPT LINE WHICH ASKS FOR THE CELL POSITIONS
    *)
   PROCEDURE REPRINTTOPLINE;
   VAR
      ACROSS : INTEGER;
   BEGIN
      GOTOXY(0, 0);
      FOR ACROSS := MINBOUND + 1 TO WIDTH - 1 DO
         IF BOARD[MINBOUND + 1][ACROSS].LOOKSLIKEITIS = DEAD THEN
            WRITE(' ')
         ELSE
            WRITE('*');
   END; (* REPRINTTOPLINE *)

BEGIN
   FINISHED := FALSE;
   POPULATION := 0;
   GOTOXY(0, 0);
   WRITE('POSITION OF CELL #', POPULATION + 1:1, ' IS :');
   WHILE NOT FINISHED DO
   BEGIN
      READLN(DOWN, ACROSS);
      IF (DOWN <= MINBOUND) OR          (* FINISH ENTERING THE PATTERN BY *)
         (DOWN >= HEIGHT) OR            (* SPECIFYING A NUMBER WHICH IS   *)
         (ACROSS <= MINBOUND) OR        (* NOT IN THE PATTERN             *)
         (ACROSS >= WIDTH) THEN
         FINISHED := TRUE
      ELSE WITH BOARD[DOWN][ACROSS] DO
      BEGIN
         LIMITS(ACROSS, DOWN);
         GOTOXY(ACROSS, DOWN);
         IF LOOKSLIKEITIS = ALIVE THEN  (* ALREADY SOMETHING THERE *)
         BEGIN                          (* SO REMOVE IT            *)
            WRITE(' ');
            LOOKSLIKEITIS := DEAD;
            POPULATION := POPULATION - 1;
         END
         ELSE                           (* ADD A NEW CELL          *)
         BEGIN
            WRITE('*');
            LOOKSLIKEITIS := ALIVE;
            POPULATION := POPULATION + 1;
         END;
         GOTOXY(0, 0);
         WRITE('POSITION OF CELL #', POPULATION + 1:1, ' IS :');
      END;
   END; (* WHILE CONTINUING TO READ POSITIONS *)
   REPRINTTOPLINE;
END; (* GETPOSITIONS *)

BEGIN          (* MAIN PROGRAM *)
   INITIALIZE;
   GETPOSITIONS;
   REPEAT
      COUNTNEIGHBORS;
      UPDATE;
   UNTIL (POPULATION = 0) OR
         ((BIRTHS = 0) AND (DEATHS = 0)) OR
         KEYPRESS;
   GOTOXY(0, 0);
   IF KEYPRESS THEN              (* HIT A CHAR WHEN BORED *)
      READ(CH)
   ELSE IF POPULATION = 0 THEN   (* ALL DIED OUT *)
      WRITELN('THIS COLONY HAS DIED.')
   ELSE                          (* CHANGES CAN'T OCCUR ANY MORE *)
      WRITE('THE PATTERN IS STABLE');
END. (* LIFE *)
```

# BLACKJACK

**The Game.** Blackjack, also known as Twenty-one, is an old and popular game. The computer acts as the dealer in this version. At the beginning of each round you place a bet and you are dealt two cards. Along with a display of your cards, the computer shows you its own *second* card. For example:

```
YOU DREW THE QUEEN OF HEARTS AND THE SEVEN OF
SPADES
THE DEALER'S UP CARD IS THE TEN OF DIAMONDS
```

Next the computer tells you the value of your hand:

```
YOUR COUNT IS 17
```

and asks:

```
HIT OR STAY?
```

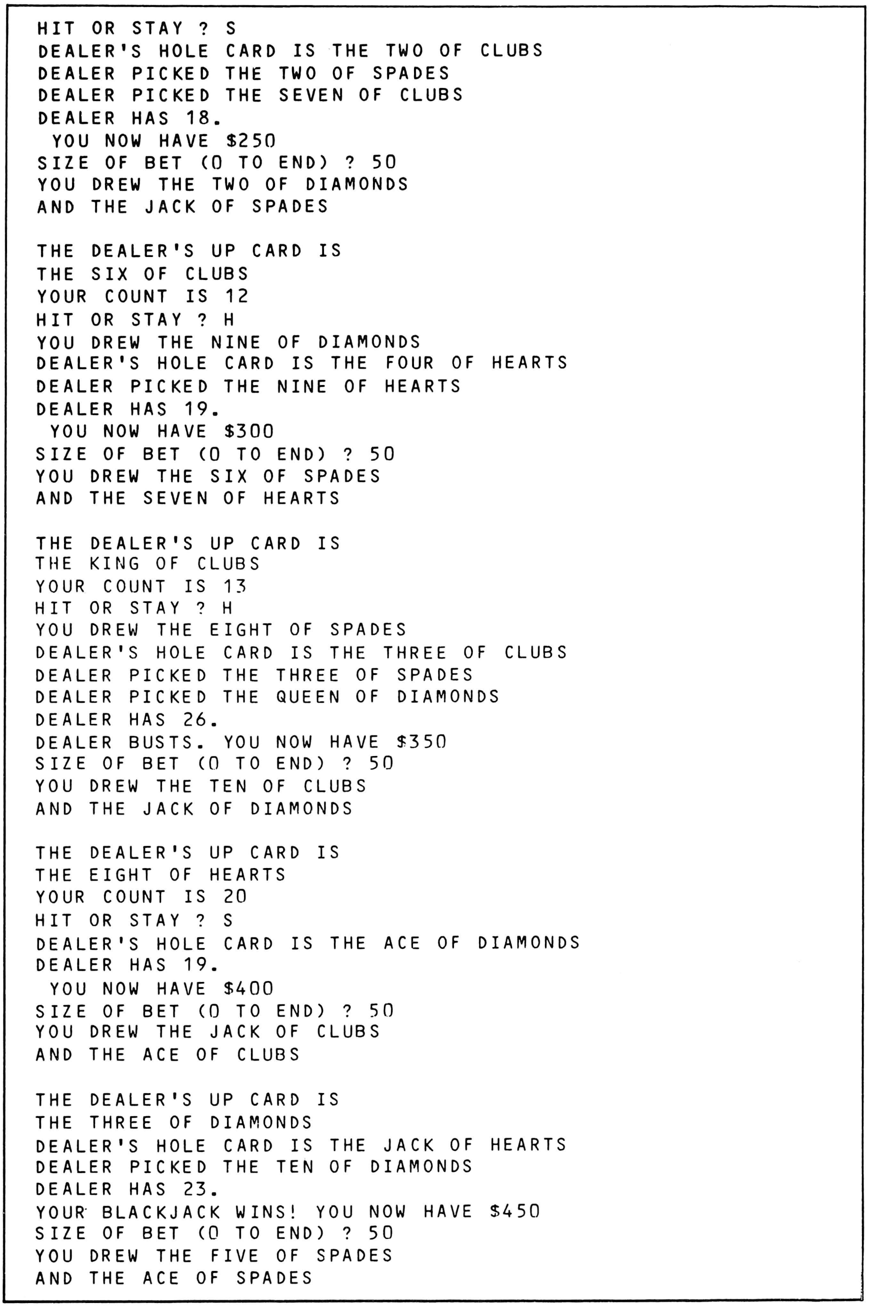
You respond with an H if you want another card or an S if you are happy with your hand as it stands. The object, of course, is to end with a hand that is as close as possible to—but not over—21. (Number cards are worth their face value; face cards are worth 10; an ace is worth either 1 or 11.) You may "hit" up to three times—for a total of five cards.

Next the computer takes its turn. Not knowing the value of your hand, the dealer "stays" with a hand that is worth 17 or better, and "hits" for 16 or less.

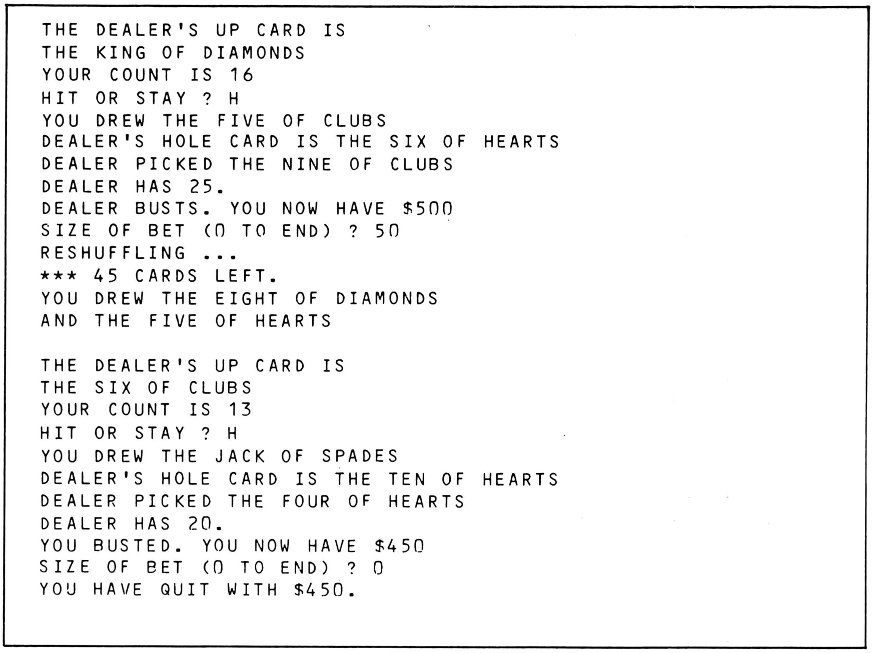
When both turns are played, the owner of the hand that is closer to 21 is the winner, and collects the bet. If you go over 21, you have "busted" and the computer automatically wins (unless it too has "busted", in which case the game is a draw). In the case of a tie, the computer collects the bet.

You begin the game with $100. The house limits on bets are $2 (minimum) to $200 (maximum). As long as your cash holds out, you can keep playing; the house offers no credit in this game.

```
                BLACKJACK FOR ONE
DO YOU WANT INSTRUCTIONS? (Y IF SO) ? Y
THIS PROGRAM PLAYS A SIMPLE VERSION
OF BLACKJACK. NEITHER SPLITTING,
NOR MODIFICATION OF THE BET AFTER
THE HAND HAS BEEN DEALT IS ALLOWED

*** 52 CARDS LEFT.
SIZE OF BET (0 TO END) ? 50
YOU DREW THE QUEEN OF HEARTS
AND THE ACE OF HEARTS

THE DEALER'S UP CARD IS
THE FOUR OF DIAMONDS
DEALER'S HOLE CARD IS THE SIX OF DIAMONDS
DEALER PICKED THE FOUR OF SPADES
DEALER PICKED THE NINE OF SPADES
DEALER HAS 23.
YOUR BLACKJACK WINS! YOU NOW HAVE $150
SIZE OF BET (0 TO END) ? 50
YOU DREW THE KING OF HEARTS
AND THE EIGHT OF CLUBS

THE DEALER'S UP CARD IS
THE FIVE OF DIAMONDS
YOUR COUNT IS 18
HIT OR STAY ? S
DEALER'S HOLE CARD IS THE SEVEN OF SPADES
DEALER PICKED THE KING OF SPADES
DEALER HAS 22.
DEALER BUSTS. YOU NOW HAVE $200
SIZE OF BET (0 TO END) ? 50
YOU DREW THE QUEEN OF CLUBS
AND THE QUEEN OF SPADES

THE DEALER'S UP CARD IS
THE SEVEN OF DIAMONDS
YOUR COUNT IS 20
```



```
HIT OR STAY ? S
DEALER'S HOLE CARD IS THE TWO OF CLUBS
DEALER PICKED THE TWO OF SPADES
DEALER PICKED THE SEVEN OF CLUBS
DEALER HAS 18.
 YOU NOW HAVE $250
SIZE OF BET (0 TO END) ? 50
YOU DREW THE TWO OF DIAMONDS
AND THE JACK OF SPADES

THE DEALER'S UP CARD IS
THE SIX OF CLUBS
YOUR COUNT IS 12
HIT OR STAY ? H
YOU DREW THE NINE OF DIAMONDS
DEALER'S HOLE CARD IS THE FOUR OF HEARTS
DEALER PICKED THE NINE OF HEARTS
DEALER HAS 19.
 YOU NOW HAVE $300
SIZE OF BET (0 TO END) ? 50
YOU DREW THE SIX OF SPADES
AND THE SEVEN OF HEARTS

THE DEALER'S UP CARD IS
THE KING OF CLUBS
YOUR COUNT IS 13
HIT OR STAY ? H
YOU DREW THE EIGHT OF SPADES
DEALER'S HOLE CARD IS THE THREE OF CLUBS
DEALER PICKED THE THREE OF SPADES
DEALER PICKED THE QUEEN OF DIAMONDS
DEALER HAS 26.
DEALER BUSTS. YOU NOW HAVE $350
SIZE OF BET (0 TO END) ? 50
YOU DREW THE TEN OF CLUBS
AND THE JACK OF DIAMONDS

THE DEALER'S UP CARD IS
THE EIGHT OF HEARTS
YOUR COUNT IS 20
HIT OR STAY ? S
DEALER'S HOLE CARD IS THE ACE OF DIAMONDS
DEALER HAS 19.
 YOU NOW HAVE $400
SIZE OF BET (0 TO END) ? 50
YOU DREW THE JACK OF CLUBS
AND THE ACE OF CLUBS

THE DEALER'S UP CARD IS
THE THREE OF DIAMONDS
DEALER'S HOLE CARD IS THE JACK OF HEARTS
DEALER PICKED THE TEN OF DIAMONDS
DEALER HAS 23.
YOUR BLACKJACK WINS! YOU NOW HAVE $450
SIZE OF BET (0 TO END) ? 50
YOU DREW THE FIVE OF SPADES
AND THE ACE OF SPADES
```



```
THE DEALER'S UP CARD IS
THE KING OF DIAMONDS
YOUR COUNT IS 16
HIT OR STAY ? H
YOU DREW THE FIVE OF CLUBS
DEALER'S HOLE CARD IS THE SIX OF HEARTS
DEALER PICKED THE NINE OF CLUBS
DEALER HAS 25.
DEALER BUSTS. YOU NOW HAVE $500
SIZE OF BET (0 TO END) ? 50
RESHUFFLING ...
*** 45 CARDS LEFT.
YOU DREW THE EIGHT OF DIAMONDS
AND THE FIVE OF HEARTS

THE DEALER'S UP CARD IS
THE SIX OF CLUBS
YOUR COUNT IS 13
HIT OR STAY ? H
YOU DREW THE JACK OF SPADES
DEALER'S HOLE CARD IS THE TEN OF HEARTS
DEALER PICKED THE FOUR OF HEARTS
DEALER HAS 20.
YOU BUSTED. YOU NOW HAVE $450
SIZE OF BET (0 TO END) ? 0
YOU HAVE QUIT WITH $450.
```

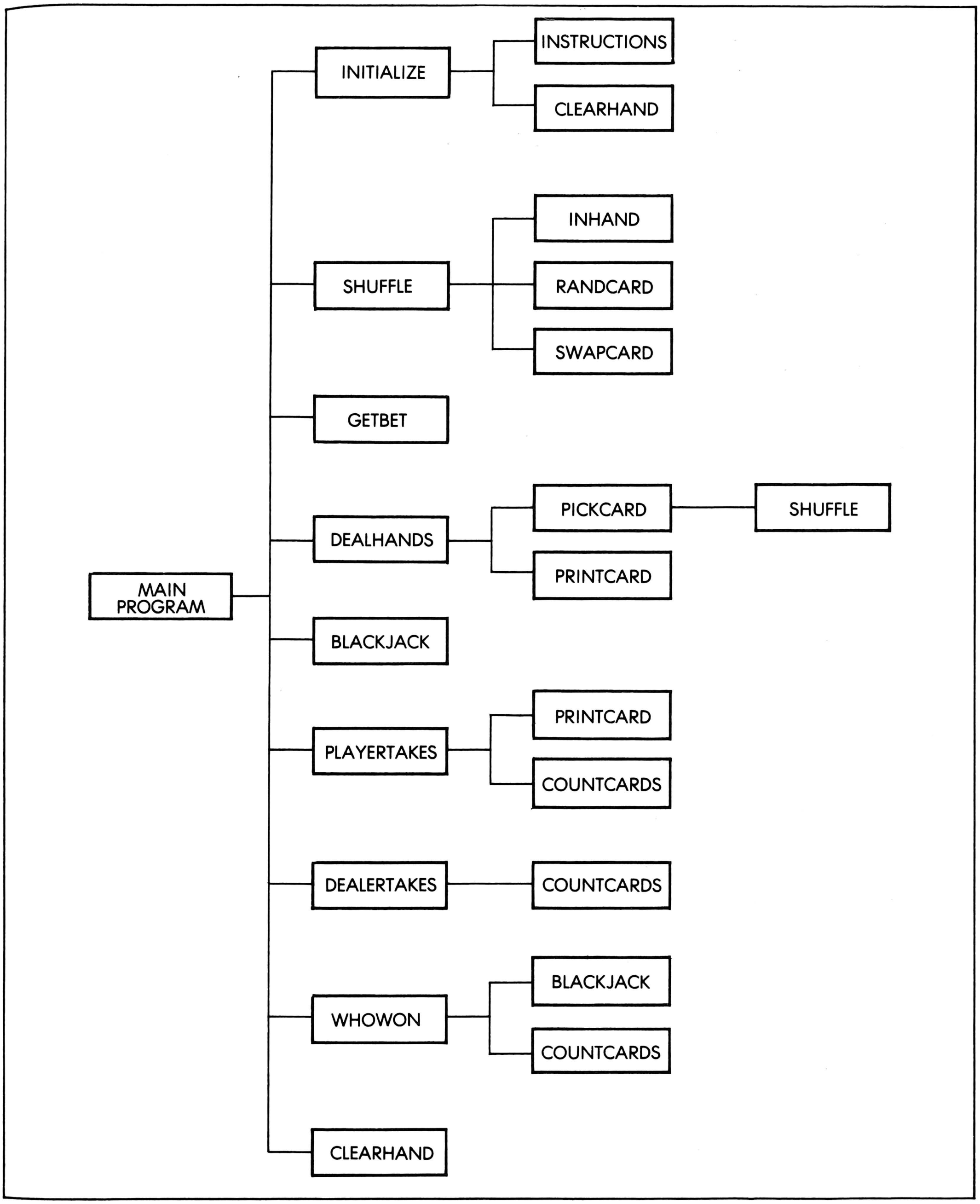
# The Program.

The main data type of this program is CARD, a record with three fields:

```
CARD = RECORD
   SUIT: CARDSUIT;
   VALUE: CARDVALUE;
   STATE: CARDSTATE;
END;
```

The third field, STATE, indicates whether a card is in a hand (PICKED) or in the deck (INDECK). DECK, PLAYER, and DEALER are all arrays of CARD.

Before the first round begins, the main program calls procedures INITIALIZE and SHUFFLE.


MAIN PROGRAM
INITIALIZE
INSTRUCTIONS
CLEARHAND
SHUFFLE
INHAND
RANDCARD
SWAPCARD
GETBET
DEALHANDS
PICKCARD
SHUFFLE
PRINTCARD
BLACKJACK
PLAYERTAKES
PRINTCARD
COUNTCARDS
DEALERTAKES
COUNTCARDS
WHOWON
BLACKJACK
COUNTCARDS
CLEARHAND

INITIALIZE begins by offering a brief description of the game (via a call to INSTRUCTIONS), and then makes two calls to CLEARHAND to initialize the STATE fields of PLAYER and DEALER:

```
FOR HANDINDEX := 1 TO MAXHANDSIZE DO
  AHAND[HANDINDEX].STATE := INDECK;
```

Then values are assigned to elements of the array COUNTVALUE (the integral count value of each card), and VALUENAME and SUITNAME (strings used to display a card).

Procedure SHUFFLE actually creates the deck (assigning values to the elements of DECK) and calls procedure SWAPCARD once for each card, to shuffle the deck. The call to SWAPCARD has, as one of its parameters, a call to function RANDCARD so that each card will be exchanged with a randomly chosen card elsewhere in the deck:

```
SWAPCARD(I,RANDCARD(CARDSLFT));
```

Since SHUFFLE may be called again later in the program (from function PICKCARD when the number of cards in the deck gets down to MINCARDS), a test is included to make sure a given card is not in one of the two hands. If a card is part of a hand, then it is not included in the deck to be shuffled:

```
IF NOT (INHAND(DECK[I],PLAYER) OR
          INHAND(DECK[I],DEALER)) THEN
BEGIN
  STATE := INDECK;
  I := I + 1;
END;
```

(Function INHAND checks the card—DECK[I]—with every element of the hand—PLAYER or DEALER—and returns a TRUE if the card is in the hand.)

The procedure calls that conduct a round of the game are all contained within a **REPEAT/UNTIL** loop in the main program. Procedure GETBET is called to read the player's bet and validate the input. (A bet of 0 ends the game via an EXIT statement.) Procedure DEALHANDS assigns the first four cards from the top of the deck to the first two elements of the arrays PLAYER and DEALER. (The calls to function PICKCARD simply increment CURCARD, the index into DECK. Recall that PICKCARD also calls SHUFFLE when the deck is low.) The player's two cards and the dealer's "up card" (DEALER[2]) are displayed via calls to PRINTCARD. (PRINTCARD merely writes the appropriate elements of VALUENAME and SUITNAME.) BLACKJACK returns a TRUE if the first two elements of the hand it is passed have a count of 21. Notice that the index that references elements of COUNTVALUE is the VALUE field of a record of type CARD:

```
COUNTVALUE[SOMEONE[1].VALUE]
```

Procedures PLAYERTAKES and DEALERTAKES deal additional cards to the hands. The player is given the chance to hit or stay, and receives the next card in the deck for a hit:

```
PLAYER[ATCARD] := DECK[PICKCARD];
```

The dealer (i.e., the computer) hits if the current count of its hand is less than the constant DEALERSTAYS (i.e., 17). Finally, procedure WHOWON compares the counts of the two hands, determines the winner, and adjusts the value of MONEY (the player's assets) accordingly.

All three of these last procedures (PLAYERTAKES, DEALERTAKES, and WHOWON) call procedure COUNTCARDS to determine the count of a hand. COUNTCARDS uses the STATE field of a hand to determine how many cards the hand actually has:

```
IF SOMEONE[HANDINDEX].STATE = PICKED THEN
   WITH SOMEONE[HANDINDEX] DO
```

The count is then determined by summing up the appropriate elements of COUNTVALUE. The number of aces in the hand is also carefully tallied (we will see why shortly):

```
BEGIN
   IF VALUE = ACE THEN
      NUMACES := NUMACES + 1;
   SUM := SUM + COUNTVALUE[VALUE];
END;
```

Now, if the count of the hand exceeds 21 (SUM > BUSTED), the count of each ace in the hand is reduced (one by one) from a value of 11 to a value of 1:

```
WHILE (NUMACES > 0) AND (SUM > BUSTED) DO
BEGIN
  NUMACES := NUMACES - 1;
  SUM := SUM - 10;
END;
```

```
(*$S+*)
PROGRAM BJGAME;
USES
   APPLESTUFF;                      (* FOR RANDOM NUMBER FUNCTIONS      *)

CONST
   DECKSIZE      = 52;              (* NUMBER OF CARDS IN A DECK        *)
   MAXHANDSIZE   = 5;               (* MAXIMUM NUMBER OF CARDS THAT A   *)
                                    (* HAND MAY CONTAIN, IF MORE THAN   *)
                                    (* THIS MANY AND NOT BUSTED HE WINS*)
   MINCARDS      = 5;               (* MINIMUM NUMBER OF CARDS ALLOWED   *)
                                    (* IN THE PLAYING DECK              *)
   DEALERSTAYS   = 17;              (* DEALER TAKES ANOTHER CARD IF     *)
                                    (* HE HAS LESS THAN THIS MANY       *)
   BUSTED        = 21;
   STARTAMOUNT   = 100;             (* MONEY PLAYER HAS AT START (HE'S *)
                                    (* SURE TO LOSE IT)                 *)
   MINBET        = 2;               (* HOUSE LIMITS ON SIZE OF LARGEST *)
   MAXBET        = 200;             (* AND SMALLEST BETS                *)
TYPE
   CARDSUIT = (SPADES, HEARTS, CLUBS, DIAMONDS);
   CARDVALUE = (DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
                NINE, TEN, JACK, QUEEN, KING, ACE);
   CARDSTATE = (PICKED, INDECK);
   CARD = RECORD
             SUIT : CARDSUIT;
             VALUE : CARDVALUE;
             STATE : CARDSTATE;
          END;
   HAND = ARRAY [1 .. MAXHANDSIZE] OF CARD;
VAR
   DECK      : ARRAY[1 .. DECKSIZE] OF CARD;
   CARDSLEFT : INTEGER;

   SUITNAME   : ARRAY[CARDSUIT] OF STRING;
   VALUENAME  : ARRAY[CARDVALUE] OF STRING;
   COUNTVALUE : ARRAY[CARDVALUE] OF INTEGER;

   PLAYER, DEALER : HAND;
   MONEY, BET     : INTEGER;
   CURCARD : INTEGER;
```

```
(*
 * WRITE THE SUIT AND VALUE OF A CARD.
 *)
PROCEDURE PRINTCARD(ACARD : CARD);
BEGIN
   WRITE('THE ', VALUENAME[ACARD.VALUE]);
   WRITELN(' OF ', SUITNAME[ACARD.SUIT]);
END;

(*
 * ASKS FOR INSTRUCTIONS
 *)

PROCEDURE INSTRUCTIONS;
VAR
   RESPONSE : CHAR;
BEGIN
   WRITELN(' ':15, 'BLACKJACK FOR ONE');
   WRITE('DO YOU WANT INSTRUCTIONS? (Y IF SO) ? ');
   READLN(RESPONSE):
   IF RESPONSE = 'Y' THEN
   BEGIN
      WRITELN('THIS PROGRAM PLAYS A SIMPLE VERSION');
      WRITELN('OF BLACKJACK. NEITHER SPLITTING,');
      WRITELN('NOR MODIFICATION OF THE BET AFTER');
      WRITELN('THE HAND HAS BEEN DEALT IS ALLOWED');
   END;
   WRITELN;
END;

(*
 * RETURNS TRUE IF THE CARD C IS IN THE HAND
 *)

FUNCTION INHAND(C : CARD; WHOSE : HAND) : BOOLEAN;
VAR
   HANDINDEX : INTEGER;
BEGIN
   FOR HANDINDEX := 1 TO MAXHANDSIZE DO
      IF ((C.SUIT = WHOSE[HANDINDEX].SUIT) AND
         (C.VALUE = WHOSE[HANDINDEX].VALUE)) THEN
      BEGIN
         INHAND := TRUE;
         EXIT(INHAND);
      END;
   INHAND := FALSE;
END; (* INHAND *)

(*
 * RETURN A RANDOM INDEX INTO THE DECK
 *)

FUNCTION RANDCARD(L:INTEGER) : INTEGER;
BEGIN
   RANDCARD := 1 + RANDOM MOD L;
END;
```

```
(*
 * REMOVES ALL CARDS FROM THE ARGUMENT HAND
 *)

PROCEDURE CLEARHAND(VAR AHAND : HAND);
VAR
   HANDINDEX : INTEGER;
BEGIN
   FOR HANDINDEX := 1 TO MAXHANDSIZE DO
      AHAND[HANDINDEX].STATE := INDECK;
END;

(*
 * INITIALIZE THE NAMES OF THE SUITS AND VALUES
 *)

PROCEDURE INITIALIZE;
VAR
    I : INTEGER;
    CARDVAL : CARDVALUE;
BEGIN
   INSTRUCTIONS;
   CLEARHAND(PLAYER);
   CLEARHAND(DEALER);
   MONEY  := STARTAMOUNT;     (* GIVE PLAYER MONEY WITH WHICH TO PLAY *)
   CARDSLEFT := 0;
   I := 2;
   FOR CARDVAL := DEUCE TO TEN DO
   BEGIN
       COUNTVALUE[CARDVAL] := I;
       I := I + 1;
   END;
   FOR CARDVAL := JACK TO KING DO
      COUNTVALUE[CARDVAL] := 10;
   COUNTVALUE[ACE] := 11;

       VALUENAME[DEUCE] := 'TWO';
       VALUENAME[THREE] := 'THREE';
       VALUENAME[FOUR]  := 'FOUR';
       VALUENAME[FIVE]  := 'FIVE';
       VALUENAME[SIX]   := 'SIX';
       VALUENAME[SEVEN] := 'SEVEN';
       VALUENAME[EIGHT] := 'EIGHT';
       VALUENAME[NINE]  := 'NINE';
       VALUENAME[TEN]   := 'TEN';
       VALUENAME[JACK]  := 'JACK';
       VALUENAME[QUEEN] := 'QUEEN';
       VALUENAME[KING]  := 'KING';
       VALUENAME[ACE]   := 'ACE';
       SUITNAME[SPADES] := 'SPADES';
       SUITNAME[HEARTS] := 'HEARTS';
       SUITNAME[CLUBS]  := 'CLUBS';
       SUITNAME[DIAMONDS] := 'DIAMONDS';
       RANDOMIZE;
END;
```

```
(*
 * SHUFFLES THE CARDS THAT ARE NOT IN
 * EITHER PLAYER'S HAND.  THE INITIAL SHUFFLE
 * DOES ALL THE CARDS BECAUSE BOTH HANDS
 * START EMPTY.
 *)
PROCEDURE SHUFFLE;
VAR
   ASUIT  : CARDSUIT;
   AVALUE : CARDVALUE;
   I      : INTEGER;

   (*
    * EXCHANGE THE CARDS AT THE TWO POSITIONS
    * IN THE DECK.
    *)
   PROCEDURE SWAPCARD(FIRST, SECOND : INTEGER);
   VAR
      TEMPCARD : CARD;
   BEGIN
      TEMPCARD := DECK[FIRST];
      DECK[FIRST] := DECK[SECOND];
      DECK[SECOND] := TEMPCARD;
   END;

BEGIN           (* SHUFFLE *)
   I := 1;
   FOR ASUIT := SPADES TO DIAMONDS DO   (* GO THROUGH EVERY POSSIBLE *)
      FOR AVALUE := DEUCE TO ACE DO     (* CARD                      *)
         WITH DECK[I] DO
         BEGIN
            SUIT := ASUIT;
            VALUE := AVALUE;
            IF NOT (INHAND(DECK[I], PLAYER) OR    (* DO NOT SHUFFLE CARDS *)
                    INHAND(DECK[I], DEALER)) THEN (* THAT ARE IN A HAND   *)
            BEGIN
               STATE := INDECK;
               I := I + 1;
            END;
         END;
   CURCARD := 0;
   CARDSLEFT := I - 1;
      WRITELN('*** ', CARDSLEFT:1, ' CARDS LEFT.');
   FOR I := 1 TO CARDSLEFT DO            (* RANDOMIZE THE DECK *)
      SWAPCARD(I,RANDCARD(CARDSLEFT));
END; (* SHUFFLE *)
(*
 * RETURNS THE INDEX INTO THE DECK OF THE NEXT CARD.
 * CALLS SHUFFLE IF DECK IS NEARLY FINISHED
 *)
FUNCTION PICKCARD : INTEGER;
BEGIN
   IF CARDSLEFT < MINCARDS THEN
   BEGIN
      WRITELN('RESHUFFLING ...');
      SHUFFLE;
   END;
   CURCARD := CURCARD + 1;
   DECK[CURCARD].STATE := PICKED;
   CARDSLEFT := CARDSLEFT - 1;
   PICKCARD := CURCARD;

END;

(*
 * DETERMINES THE SUM OF THE VALUES IN A HAND.
 * A CARD'S STATE MUST BE 'PICKED' FOR IT TO BE INCLUDED.
 * ACES ARE ASSUMED TO BE 11. IF THE COUNT IS OVER 21 AND
 * THERE ARE ACES IN IT AS MANY AS ARE NEEDED WILL BE
 * DEVALUED TO 1.
 *)
FUNCTION COUNTCARDS (SOMEONE : HAND) : INTEGER;
VAR
   HANDINDEX, SUM, NUMACES : INTEGER;
BEGIN
   SUM := 0;
   NUMACES := 0;
   FOR HANDINDEX := 1 TO MAXHANDSIZE DO
      IF SOMEONE[HANDINDEX].STATE = PICKED THEN (* A CARD IS REALLY THERE *)
         WITH SOMEONE[HANDINDEX] DO
         BEGIN
            IF VALUE = ACE THEN
               NUMACES := NUMACES + 1;
            SUM := SUM + COUNTVALUE[VALUE];
         END;
   WHILE (NUMACES > 0) AND (SUM > BUSTED) DO    (* IF NECESSARY REDUCE   *)
   BEGIN                                        (* COUNT BY CHANGING     *)
      NUMACES := NUMACES - 1;                   (* ACES TO 1 FROM 11     *)
      SUM := SUM - 10;
   END;
   COUNTCARDS := SUM;
END;

(*
 * RETURNS TRUE IF THE ARGUMENT HAND IS A BLACKJACK
 *)
FUNCTION BLACKJACK(SOMEONE : HAND) : BOOLEAN;
BEGIN
   BLACKJACK := ((COUNTVALUE[SOMEONE[1].VALUE] = 10) AND
                 (COUNTVALUE[SOMEONE[2].VALUE] = 11)) OR
                ((COUNTVALUE[SOMEONE[1].VALUE] = 11) AND
                 (COUNTVALUE[SOMEONE[2].VALUE] = 10));
END;

PROCEDURE GETBET;
CONST
   BETPROMPT = 'SIZE OF BET (0 TO END) ? ';
BEGIN
   WRITE(BETPROMPT);
   READLN(BET);
   WHILE NOT (BET IN [0, MINBET .. MAXBET]) OR
         (BET > MONEY) DO
   BEGIN
      WRITE('A BET MUST BE BETWEEN ');
      WRITELN(MINBET:1, ' AND ', MAXBET:1);
      WRITELN('AND MUST BE NO LARGER THAN THE AMOUNT');
      WRITELN('OF MONEY YOU HAVE. ENTER 0 TO LEAVE');
      WRITE(BETPROMPT);
      READLN(BET);
   END;
   IF BET = 0 THEN
   BEGIN
      WRITELN('YOU HAVE QUIT WITH $', MONEY:1, '.');
      EXIT(BJGAME);
   END;
END;
(*
 * DEALS THE CARDS TO BOTH PARTICIPANTS FOR
 * THIS HAND.
 *)

PROCEDURE DEALHANDS;
BEGIN
   PLAYER[1] := DECK[PICKCARD];
   DEALER[1] := DECK[PICKCARD];
   PLAYER[2] := DECK[PICKCARD];
   DEALER[2] := DECK[PICKCARD];
   WRITE('YOU DREW ');
   PRINTCARD(PLAYER[1]);
   WRITE('AND ');
   PRINTCARD(PLAYER[2]);
   WRITELN;
   WRITELN('THE DEALER''S UP CARD IS ');
   PRINTCARD(DEALER[2]);
END;

(*
 * ASKS THE PLAYER IF MORE CARDS ARE WANTED.
 *)
PROCEDURE PLAYERTAKES;
VAR
   ATCARD : INTEGER;
   ANSWER : CHAR;

BEGIN
   ATCARD := 3;
   WHILE (ATCARD <= MAXHANDSIZE) AND
         (COUNTCARDS(PLAYER) < BUSTED) DO
   BEGIN
      WRITELN('YOUR COUNT IS ', COUNTCARDS(PLAYER));
      WRITE('HIT OR STAY ? ');
      READLN(ANSWER);
      IF ANSWER = 'H' THEN
      BEGIN
         PLAYER[ATCARD] := DECK[PICKCARD];
         WRITE('YOU DREW ');
         PRINTCARD(PLAYER[ATCARD]);
         ATCARD := ATCARD + 1;
      END
      ELSE IF ANSWER = 'S' THEN
         EXIT(PLAYERTAKES)
      ELSE
         WRITELN('PLEASE ANSWER H OR S.');
  END;
  IF COUNTCARDS(PLAYER) < BUSTED THEN
     WRITELN('YOU CAN TAKE ONLY ', MAXHANDSIZE:1, ' CARDS.');
END;

PROCEDURE DEALERTAKES;
VAR
   ATCARD : INTEGER;
BEGIN
   WRITE('DEALER''S HOLE CARD IS ');
   PRINTCARD(DEALER[1]);
   ATCARD := 3;
   WHILE (ATCARD <= MAXHANDSIZE) AND
         (COUNTCARDS(DEALER) < DEALERSTAYS) DO
   BEGIN
      DEALER[ATCARD] := DECK[PICKCARD];
      WRITE('DEALER PICKED ');
      PRINTCARD(DEALER[ATCARD]);
      ATCARD := ATCARD + 1;
   END;
END;

(*
 * DETERMINES WHO WON AND ADJUSTS THE PLAYER'S
 * MONEY APPROPRIATELY.
 *)

PROCEDURE WHOWON;
BEGIN
   WRITELN('DEALER HAS ', COUNTCARDS(DEALER):1, '.');
   IF BLACKJACK(DEALER) THEN
   BEGIN
      WRITE('DEALER GOT A BLACKJACK.');
      MONEY := MONEY - BET;
   END
   ELSE IF BLACKJACK(PLAYER) THEN
   BEGIN
      WRITE('YOUR BLACKJACK WINS!');
      MONEY := MONEY + BET;
   END
   ELSE IF COUNTCARDS(PLAYER) > BUSTED THEN
   BEGIN
      WRITE('YOU BUSTED.');
      IF COUNTCARDS(DEALER) > BUSTED THEN
         WRITE(' SO DID THE DEALER. NO PAYOUT')
      ELSE
         MONEY := MONEY - BET;
   END
   ELSE IF COUNTCARDS(DEALER) > BUSTED THEN
   BEGIN
      WRITE('DEALER BUSTS.');
      MONEY := MONEY + BET;
   END
   ELSE IF COUNTCARDS(DEALER) >= COUNTCARDS(PLAYER) THEN
      MONEY := MONEY - BET
   ELSE
      MONEY := MONEY + BET;
   WRITELN(' YOU NOW HAVE $', MONEY:1);
END;

BEGIN           (* BLACKJACK *)
   INITIALIZE;
   SHUFFLE;
   REPEAT
      GETBET;
      DEALHANDS;
      IF NOT BLACKJACK(PLAYER) THEN
         PLAYERTAKES;
      DEALERTAKES;
      WHOWON;
      CLEARHAND(PLAYER);
      CLEARHAND(DEALER);
   UNTIL MONEY <= 0;
   WRITELN('YOU HAVE RUN OUT OF MONEY');
END.
```

WUMPUS

# WUMPUS

**The Game.** Enter the cave of the Wumpus, where danger and adventure await you. The cave is a maze of twenty interconnected rooms. You will travel from room to room hunting for the Wumpus; with the right combination of skill and luck, you might destroy the creature with one of your seven specially designed Wumpus-killing arrows. But beware: every step of your trip through the cave is a potential disaster. In some rooms lurk sinister bats, who are ready to snatch you up and fly you at their whim to another room. Some rooms contain terrifying pits in which you will certainly perish if you should fall. But the worst of all is the ravenous Wumpus himself, who delights in devouring mystified humans ensnared in his cave.

Each time you move to a new room, you will receive a message like this one:

```
YOU ARE IN ROOM 17
THERE ARE TUNNELS TO ROOMS
   9 12 13
```

In addition, you may see one or more of the following messages to warn you of nearby perils:

```
I SMELL A WUMPUS.
I HEAR BATS.
I FEEL A DRAFT.
```

You can move by typing an M and the number of one of the rooms connected to your current location. For example:

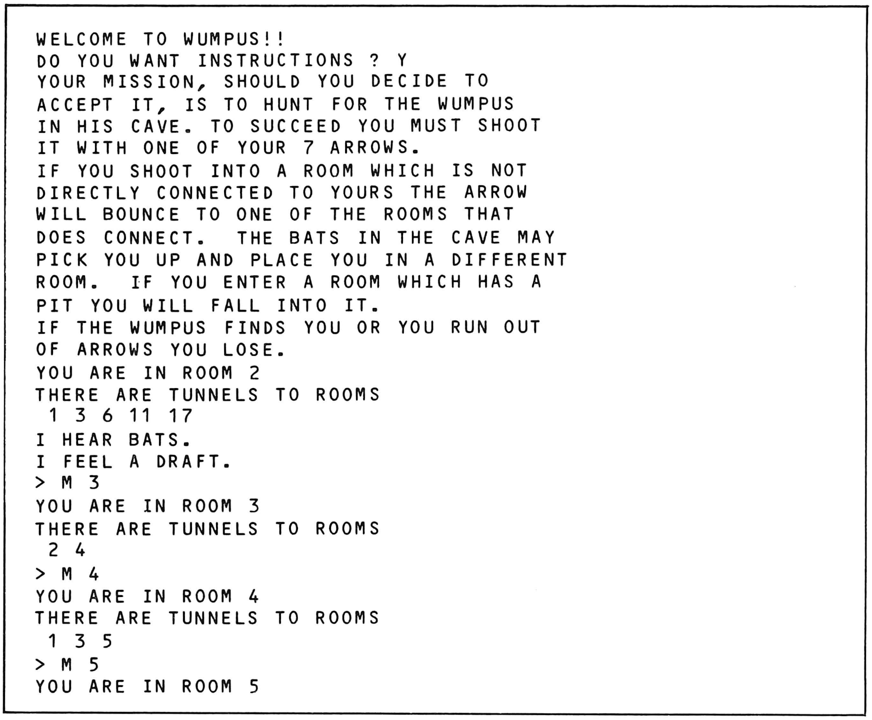
M 12

You can shoot an arrow into a room of the cave by typing an S and the room number:

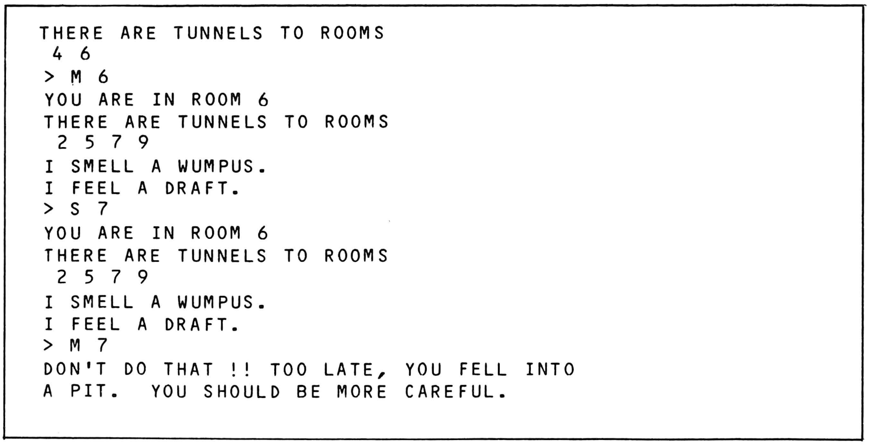
S 9

If you get confused (the steamy, poisoned air of the Wumpus's cave tends to distract even the most clear-thinking hunter) type a question mark (?) for instructions.

Finally, if the strain is just too great and you cannot continue your perilous mission, type a Q to leave the cave and retreat to the relative safety of the computer room.



```
WELCOME TO WUMPUS!!
DO YOU WANT INSTRUCTIONS ? Y
YOUR MISSION, SHOULD YOU DECIDE TO
ACCEPT IT, IS TO HUNT FOR THE WUMPUS
IN HIS CAVE. TO SUCCEED YOU MUST SHOOT
IT WITH ONE OF YOUR 7 ARROWS.
IF YOU SHOOT INTO A ROOM WHICH IS NOT
DIRECTLY CONNECTED TO YOURS THE ARROW
WILL BOUNCE TO ONE OF THE ROOMS THAT
DOES CONNECT.  THE BATS IN THE CAVE MAY
PICK YOU UP AND PLACE YOU IN A DIFFERENT
ROOM.  IF YOU ENTER A ROOM WHICH HAS A
PIT YOU WILL FALL INTO IT.
IF THE WUMPUS FINDS YOU OR YOU RUN OUT
OF ARROWS YOU LOSE.
YOU ARE IN ROOM 2
THERE ARE TUNNELS TO ROOMS
 1 3 6 11 17
I HEAR BATS.
I FEEL A DRAFT.
> M 3
YOU ARE IN ROOM 3
THERE ARE TUNNELS TO ROOMS
 2 4
> M 4
YOU ARE IN ROOM 4
THERE ARE TUNNELS TO ROOMS
 1 3 5
> M 5
YOU ARE IN ROOM 5
```



```
THERE ARE TUNNELS TO ROOMS
 4 6
> M 6
YOU ARE IN ROOM 6
THERE ARE TUNNELS TO ROOMS
 2 5 7 9
I SMELL A WUMPUS.
I FEEL A DRAFT.
> S 7
YOU ARE IN ROOM 6
THERE ARE TUNNELS TO ROOMS
 2 5 7 9
I SMELL A WUMPUS.
I FEEL A DRAFT.
> M 7
DON'T DO THAT !! TOO LATE, YOU FELL INTO
A PIT.  YOU SHOULD BE MORE CAREFUL.
```

# The Program.

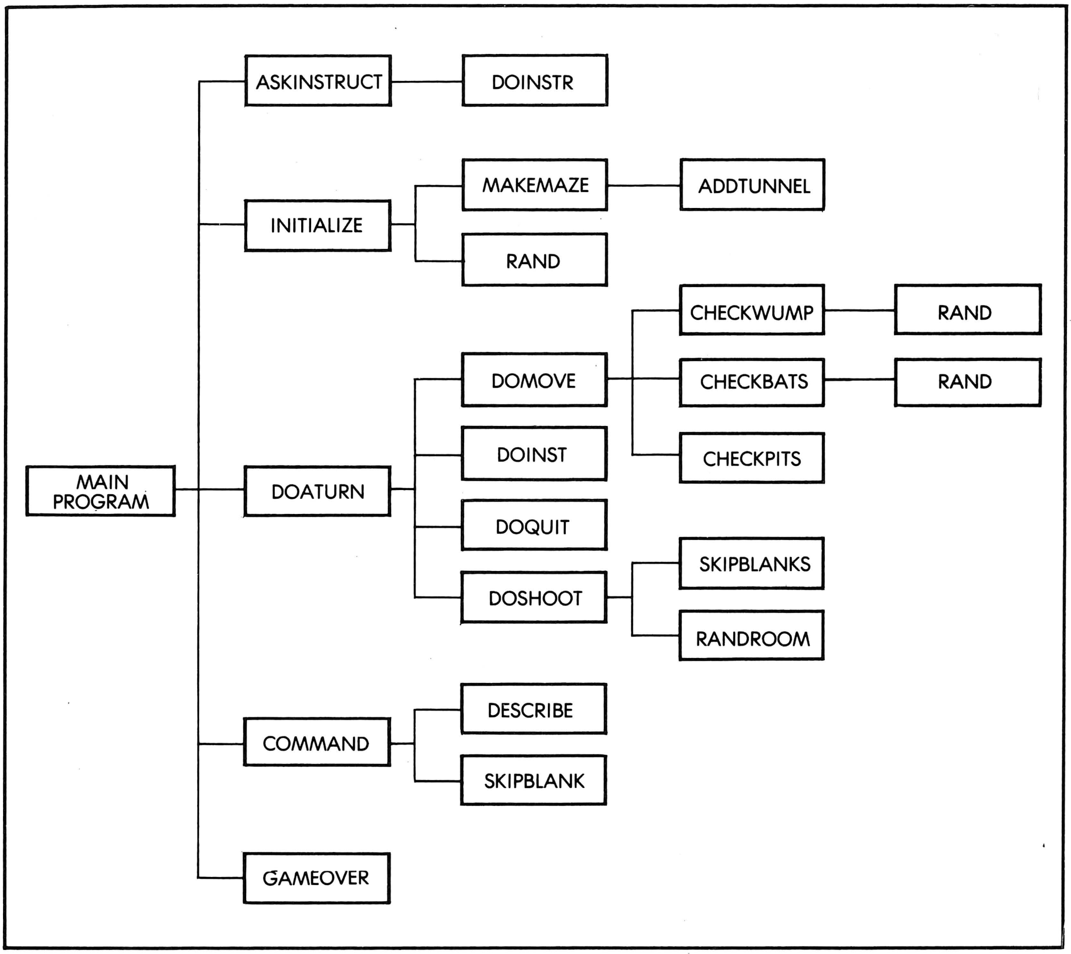
Program HUNTTHEWUMPUS makes clear and effective use of sets to keep track of the potentially confusing events inside the Wumpus's cave. The set type ROOMS is defined in the type declaration section:

```
TYPE
   ROOM = 1..MAXROOMS;
   ROOMS = SET OF ROOM;
```

In the **VAR** section, a number of variables are defined in terms of ROOM and ROOMS:

```
VAR
   CAVE: ARRAY[ROOM] OF ROOMS;
   PLAYER: ROOM;
   WUMPUS: ROOM;
   BATS: ROOMS;
   PITS: ROOMS;
```

Each element of CAVE is thus a set representing one of the 20 rooms; the elements of each set represent the rooms to which a given


MAIN PROGRAM
ASKINSTRUCT
DOINSTR
INITIALIZE
MAKEMAZE
ADDTUNNEL
RAND
DOATURN
DOMOVE
CHECKWUMP
RAND
CHECKBATS
RAND
CHECKPITS
DOINST
DOQUIT
DOSHOOT
SKIPBLANKS
RANDROOM
COMMAND
DESCRIBE
SKIPBLANK
GAMEOVER

room is connected by tunnels. PLAYER and WUMPUS are integers indicating the current room location of the two main characters of the game. BATS and PITS are sets whose elements represent the room numbers of the bats and pits.

As usual, the only problem with using sets is that their elements cannot be referenced individually. Writing the elements of a set is thus a rather clumsy procedure. However, as we will see in this program, the special set "operations" (including **IN**, *, and +, for set inclusion, intersection, and union respectively) make the use of sets very attractive in a program that requires many comparisons of groups of elements.

After a display of the instructions (via calls to ASKINSTRUCT and DOINSTR) the main program calls procedure INITIALIZE to set the scene for the game.

INITIALIZE begins by setting BATS, PITS, and all the elements of CAVE to the null set:

```
FOR I := 1 TO MAXROOMS DO
  CAVE[I] := [ ];
BATS := [ ];
PITS := [ ];
```

Then procedure MAKEMAZE is called to create the interconnected rooms of the cave.

MAKEMAZE has two **FOR** loops. The first loop connects each room to the room directly after it in numerical order. Calls to ADDTUNNEL actually add elements to each set to represent the tunnel connections. ADDTUNNEL contains two statements; to make a connection between rooms, each set must contain the number of the connected room:

```
CAVE[FROM] := CAVE[FROM] + [DEST];
CAVE[DEST] := CAVE[DEST] + [FROM];
```

Notice that ADDTUNNEL is passed the numbers of the two rooms that are to be connected; these two values are received in the formal parameters FROM and DEST.

The second **FOR** loop of procedure MAKEMAZE creates random connections between rooms. Function RAND returns a random room number, and ADDTUNNEL is called to make the connection.

After the call to MAKEMAZE, procedure INITIALIZE performs several other important tasks. The locations of the Wumpus, the bats, and the pits are determined via calls to function RAND. The player's initial location is also determined randomly, except that the player must be placed in an empty room:

```
REPEAT
   PLAYER := RAND(1,MAXROOMS);
UNTIL (PLAYER <> WUMPUS) AND
         NOT (PLAYER IN PITS) AND
         NOT (PLAYER IN BATS);
```

The BOOLEAN variables QUITTING, KILLED and WUMPUSKILLED are set to FALSE; the player is given NUMBEROFARROWS (the value assigned to ARROWSLEFT); and the set COMMANDSET is assigned:

```
COMMANDSET := [MOVE, SHOOT, QUIT, HELP];
```

Notice that all four of the values in this set are constants that were defined in the program heading:

```
MOVE = 'M';
QUIT = 'Q';
SHOOT = 'S';
HELP = '?';
```

The game itself is controlled through repeated calls to procedure DOATURN (from the main program) until function GAMEOVER returns a TRUE:

```
GAMEOVER := QUITTING OR KILLED OR
                WUMPUSKILLED OR (ARROWLEFT = 0);
```

The call to DOATURN has a call to function COMMAND as its parameter:

```
DOATURN(COMMAND);
```

COMMAND in turn calls procedure DESCRIBE to display the player's location and to give messages of warning.

Procedure DESCRIBE illustrates what must be done to write the elements of a set. A **FOR** loop tests each value in the *range* of the set for actual inclusion in the set. If a value of the range is **IN** the set, then that value is written:

```
FOR I := 1 TO MAXROOMS DO
   IF I IN CAVE[PLAYER] THEN
      WRITE(' ', I:1);
```

This loop thus displays the rooms that are connected to the player's present location. The **IF** statements that follow this **FOR** loop illustrate the use of the set intersection operation (*); for example:

```
IF ((CAVE[PLAYER] * BATS) <> [ ]) THEN
   WRITELN('I HEAR BATS.');
```

The intersection of the sets CAVE[PLAYER] and BATS represents any rooms that:

1. are connected to the player's current room, *and*,
2. contain a bat.

If this intersection is *not* the null set, then at least one bat is located one room away from the player, and an appropriate message is displayed.

Back in function COMMAND, the prompt (>) is displayed and the program waits for the player to type in a command character. (Procedure SKIPBLANKS uses the buffer variable INPUT^ and the GET procedure to skip by any spaces in the input line.)

Procedure DOATURN uses the command character as the "selector" in a **CASE** statement that calls one of four procedures: DOMOVE, DOSHOOT, DOQUIT, or DOINSTR.

DOMOVE reads the number of the room that the player wants to move to, validates the input, moves the player, and calls procedures CHECKWUMP, CHECKPITS, and CHECKBATS to investigate the room that the player has moved into. CHECKWUMP possibly moves the WUMPUS to a new room (if a call to RAND returns a room number that is in CAVE[WUMPUS]). CHECKWUMP and CHECKPITS both set KILLED to TRUE (ending the game) if:

```
WUMPUS = PLAYER
```

or

```
PLAYER IN PITS
```

Procedure CHECKBATS calls RAND to determine a new room to fly the player to. However, the bat will not deposit the player in a room that has a pit, another bat, or the Wumpus in it. (Bats are capricious, but not cruel.) Notice the use of the set union operation (+) in this statement:

```
REPEAT
  FLEWTO := RAND(1,MAXROOMS)
UNTIL (NOT (FLEWTO IN (BATS + PITS))) AND
      (FLEWTO <> WUMPUS);
```

Finally, DOSHOOT reads the number of the room (NEXTROOM) that the player wants to try shooting an arrow into. If the WUMPUS is in the room, then the game ends with the player victorious:

```
IF WUMPUS = NEXTROOM THEN
  WUMPUSKILLED := TRUE
```

DOSHOOT also checks to make sure the player has not accidentally shot himself:

```
ELSE IF PLAYER = NEXTROOM THEN
  KILLLED = TRUE;
```

Otherwise the arrow is wasted (ARROWSLEFT is decremented), and the game continues.

```
 PROGRAM HUNTTHEWUMPUS (INPUT,OUTPUT,KEYBOARD);

USES    APPLESTUFF;             (* FOR RANDOM *)

CONST
        MAXROOMS = 20;
        MAXBATS  = 2;           (* AT MOST THIS MANY, MIGHT BE LESS    *)
        MAXPITS  = 2;
        NUMBEROFARROWS = 7;
        PROMPT   = '> ';
        TUNNELSPERROOM = 2;
        MOVE     = 'M';
        QUIT     = 'Q';
        SHOOT    = 'S';
        HELP     = '?';

TYPE
        ROOM = 1 .. MAXROOMS;
        ROOMS = SET OF ROOM;

VAR
        CAVE   : ARRAY[ROOM] OF ROOMS;
        PLAYER : ROOM;         (* PLAYER'S CURRENT ROOM *)
        WUMPUS : ROOM;         (* WUMPUS'S ROOM         *)
        ARROWSLEFT : INTEGER;
        QUITTING   : BOOLEAN;
        KILLED     : BOOLEAN;
        WUMPUSKILLED : BOOLEAN;
        BATS   : ROOMS;
        PITS   : ROOMS;
        COMMANDSET : SET OF CHAR;

(*
 * RAND         RETURNS A RANDOM NUMBER BETWEEN LO AND HIGH
 *)

FUNCTION RAND(LOW, HIGH : INTEGER) : INTEGER;
BEGIN
        RAND := LOW + RANDOM MOD ( HIGH - LOW + 1);
END;

PROCEDURE DOINSTR;
BEGIN
        WRITELN('YOUR MISSION, SHOULD YOU DECIDE TO ');
        WRITELN('ACCEPT IT, IS TO HUNT FOR THE WUMPUS');
        WRITELN('IN HIS CAVE. TO SUCCEED YOU MUST SHOOT');
        WRITELN('IT WITH ONE OF YOUR ', NUMBEROFARROWS:1,' ARROWS.');
        WRITELN('IF YOU SHOOT INTO A ROOM WHICH IS NOT');
        WRITELN('DIRECTLY CONNECTED TO YOURS THE ARROW');
        WRITELN('WILL BOUNCE TO ONE OF THE ROOMS THAT');
        WRITELN('DOES CONNECT.  THE BATS IN THE CAVE MAY');
        WRITELN('PICK YOU UP AND PLACE YOU IN A DIFFERENT');
        WRITELN('ROOM.  IF YOU ENTER A ROOM WHICH HAS A');
        WRITELN('PIT YOU WILL FALL INTO IT.');
        WRITELN('IF THE WUMPUS FINDS YOU OR YOU RUN OUT');
        WRITELN('OF ARROWS YOU LOSE.');
END;
```

```
PROCEDURE ASKINSTRUCT;
VAR
        ANSWER : CHAR;
BEGIN
        WRITE ('DO YOU WANT INSTRUCTIONS ? ');
        READLN(ANSWER);
        WHILE (ANSWER <> 'Y') AND (ANSWER <> 'N') DO
        BEGIN
                WRITELN('PLEASE ANSWER YES OR NO');
                WRITELN('WOULD YOU LIKE INSTRUCTIONS?');
                READLN(ANSWER);
        END;
        IF ANSWER = 'Y' THEN
                DOINSTR;
END; (* ASKINSTRUCT *)
(*
 * ADDTUNNEL       MAKE A CONNECTION BETWEEN TWO ROOMS.
 *)
PROCEDURE ADDTUNNEL(FROM, DEST : ROOM);
BEGIN
        CAVE[FROM] := CAVE[FROM] + [DEST];
        CAVE[DEST] := CAVE[DEST] + [FROM];
END; (* ADDTUNNEL *)
(*
 * MAKEMAZE       MAKES A REASONABLY RANDOM MAZE.
 *                FOR EACH ROOM TRIES TO MAKE 3 NEW TUNNELS.
 *                IF A TUNNEL ALREADY EXISTS IN THAT DIRECTION
 *                ANOTHER DIGGING THAT WAY IS NOT MADE.
 *)

PROCEDURE MAKEMAZE;
VAR
        CURRENTROOM, TUNNELTO, NEWTUNNEL : ROOM;
BEGIN
        FOR CURRENTROOM := 2 TO MAXROOMS DO
                ADDTUNNEL(CURRENTROOM, CURRENTROOM - 1);
        FOR CURRENTROOM := 3 TO MAXROOMS DO
        BEGIN
                NEWTUNNEL := RAND(1, CURRENTROOM - 1);
                IF NOT (NEWTUNNEL IN CAVE[CURRENTROOM]) THEN
                        ADDTUNNEL(CURRENTROOM, NEWTUNNEL);
        END;
END; (* MAKEMAZE *)

(*
 * SKIPBLANKS     SKIPS SPACES, TABS UNTIL EOLN
 *)

PROCEDURE SKIPBLANKS;
BEGIN
        WHILE (NOT EOLN) AND (INPUT^ <> ' ') DO
                IF EOF THEN
                        EXIT (PROGRAM)
                ELSE
                        GET(INPUT);
END; (* SKIPBLANKS *)
```

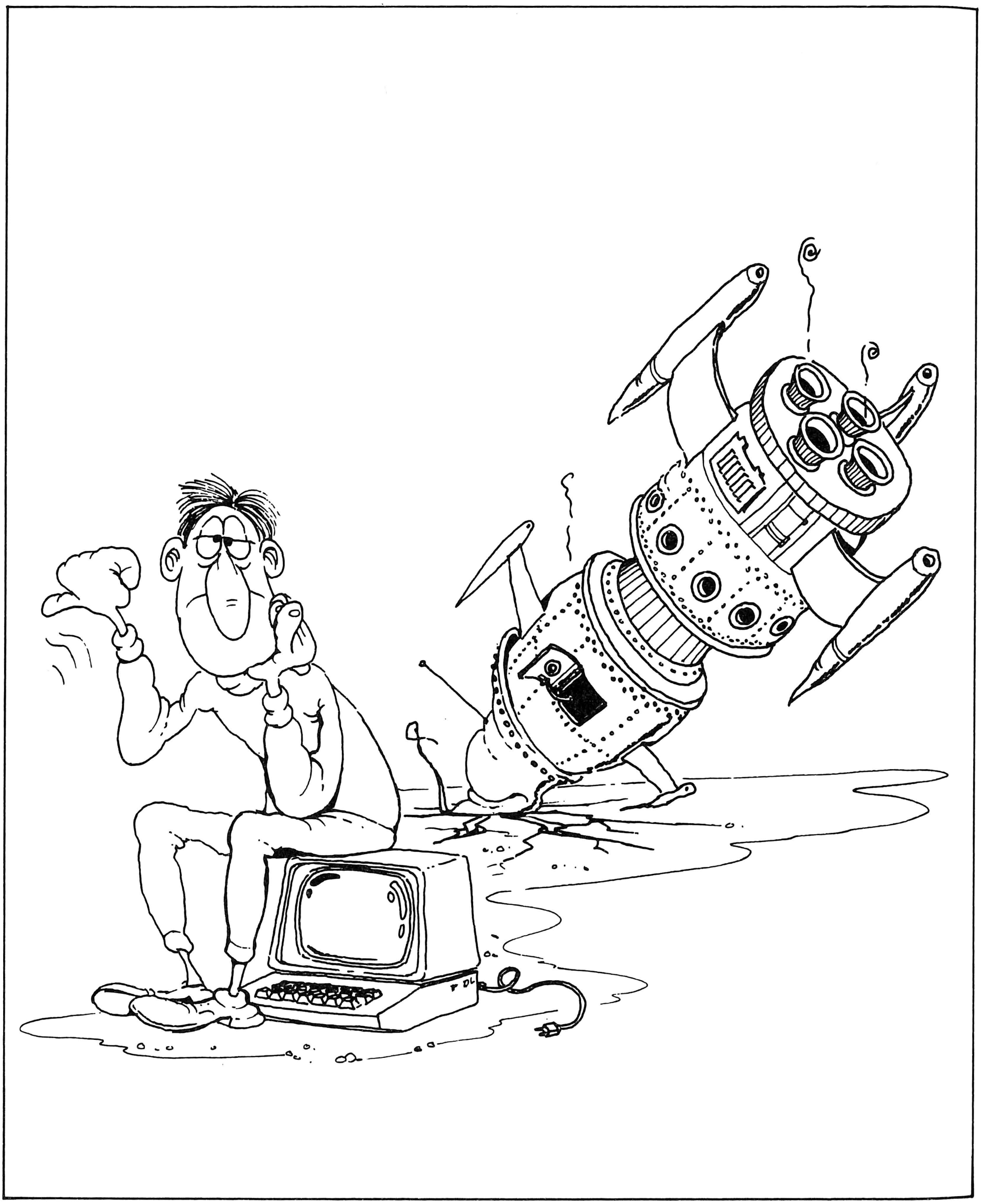
```
(*
 * DESCRIBE       GIVE A DESCRIPTION OF THE CURRENT
 *                ROOM ( PLAYER ). TELL PLAYER IF THE WUMPUS IS NEARBY
 *
 *)

PROCEDURE DESCRIBE;
VAR
        I : ROOM;

BEGIN
        WRITELN('YOU ARE IN ROOM ', PLAYER:1);
        WRITELN ('THERE ARE TUNNELS TO ROOMS');
        FOR I := 1 TO MAXROOMS DO
                IF I IN CAVE[PLAYER] THEN
                        WRITE(' ', I:1);
        WRITELN;
        IF (PLAYER IN CAVE[WUMPUS]) OR
           ((CAVE[PLAYER] * CAVE[WUMPUS]) <> []) THEN
                WRITELN('I SMELL A WUMPUS.');
        IF ((CAVE[PLAYER] * BATS) <> []) THEN
                WRITELN('I HEAR BATS.');
        IF ((CAVE[PLAYER] * PITS) <> []) THEN
                WRITELN('I FEEL A DRAFT.');
END; (* DESCRIBE *)

(*
 * COMMAND        RETURNS THE SINGLE CHARACTER THAT
 *                SIGNIFIES WHAT COMMAND WILL BE DONE.
 *)

FUNCTION COMMAND:CHAR;
VAR
        CH : CHAR;

BEGIN
        DESCRIBE;
        REPEAT
                REPEAT
                        WRITE(PROMPT);
                        SKIPBLANKS;
                        IF EOLN THEN
                                WRITE(PROMPT);
                UNTIL NOT EOLN;
                READ(CH);
                IF NOT (CH IN COMMANDSET) THEN
                BEGIN
                        READLN;
                        WRITELN(' TYPE ? FOR INSTRUCTIONS.')
                END;
        UNTIL CH IN COMMANDSET;
        COMMAND := CH;
END; (* COMMAND *)
```

```
(*
 * CHECKWUMP      MOVE THE WUMPUS, AND SEE IF IT WENT
 *                TO THE SAME ROOM AS THE PLAYER, IF SO
 *                HE'S DEAD.
 *)
PROCEDURE CHECKWUMP;
VAR
        NEWWUMPROOM : ROOM;

BEGIN
        NEWWUMPROOM := RAND(1, MAXROOMS);             (* MOVES ONLY TO   *)
        IF (NEWWUMPROOM IN CAVE[WUMPUS]) THEN         (* ADJACENT ROOMS, *)
                WUMPUS := NEWWUMPROOM;                (* OTHERWISE STAYS *)
        IF (WUMPUS = PLAYER) THEN
        BEGIN
                WRITELN('LOOK OUT !! THE WUMPUS GOT YOU.');
                WRITELN('BETTER LUCK NEXT TIME.');
                KILLED := TRUE;
        END;
END; (* CHECKWUMP *)

(*
 * CHECKBATS      IF THE PLAYER IS IN A ROOM WITH BATS THEY WILL PICK HIM
 *                UP AND MOVE HIM TO ANOTHER ROOM (WHICH WILL NOT
 *                HAVE ANY BATS IN IT).
 *)
PROCEDURE CHECKBATS;
VAR
        FLEWTO: ROOM;
BEGIN
        IF PLAYER IN BATS THEN
        BEGIN
                REPEAT
                        FLEWTO := RAND(1, MAXROOMS)
                UNTIL (NOT (FLEWTO IN (BATS+PITS))) AND
                      (FLEWTO <> WUMPUS);
                WRITELN('A SUPERBAT PICKED YOU UP AND CARRIED YOU OFF.');
                PLAYER := FLEWTO;
        END;
END; (* CHECKBATS *)

(*
 * CHECKPITS      DETERMINES IF THE PLAYER FELL INTO A PIT.
 *)

PROCEDURE CHECKPITS;
BEGIN
        IF NOT KILLED AND       (* DON'T DO IF PLAYER ALREADY DEAD *)
           (PLAYER IN PITS) THEN
        BEGIN
                WRITELN('DON''T DO THAT !! TOO LATE, YOU FELL INTO ');
                WRITELN('A PIT.  YOU SHOULD BE MORE CAREFUL.');
                KILLED := TRUE;
        END;
END; (* CHECKPITS *)
```

```
(*
 * RANDROOM      RETURNS A RANDOM ROOM NUMBER IN THE RANGE
 *               LIMITED BY THE (SET) ARGUMENT.
 *)

FUNCTION RANDROOM ( LIMITEDTO : ROOMS ) : INTEGER;
VAR
        APOSSIBILITY : ROOM;

BEGIN
        REPEAT
                APOSSIBILITY := RAND(1, MAXROOMS);
        UNTIL APOSSIBILITY IN LIMITEDTO;
        RANDROOM := APOSSIBILITY;
END; (* RANDROOM *)

(*
 * DOSHOOT       PLAYER TRIES TO SHOOT THE WUMPUS
 *               BY LISTING THE ROOMS THAT HE WANTS TO SHOOT THROUGH.
 *               IF THE ROOMS DO NOT MATCH THE LIST THE ARROW BOUNCES
 *               RANDOMLY TO A CONNECTING TUNNEL.
 *)

PROCEDURE DOSHOOT;
VAR
        NEXTROOM, LASTROOM : ROOM;

BEGIN
        SKIPBLANKS;
        LASTROOM := PLAYER;
        WHILE NOT EOLN DO
        BEGIN
                READ(NEXTROOM);
                IF WUMPUS = NEXTROOM THEN
                        WUMPUSKILLED := TRUE
                ELSE IF PLAYER = NEXTROOM THEN
                        KILLED := TRUE;
                IF NOT (NEXTROOM IN CAVE[LASTROOM]) THEN
                        NEXTROOM := RANDROOM(CAVE[LASTROOM]);
                LASTROOM := NEXTROOM;
                SKIPBLANKS;

        END;
        ARROWSLEFT := ARROWSLEFT - 1;
        IF KILLED THEN
                WRITELN('YOU KLUTZ ! YOU JUST SHOT YOURSELF.')
        ELSE IF WUMPUSKILLED THEN
                WRITELN('CONGRATULATIONS!  YOU SLEW THE FEARSOME WUMPUS')
        ELSE IF ARROWSLEFT = 0 THEN
                WRITELN('YOU RAN OUT OF ARROWS.')
END; (* DOSHOOT *)

 (*
 * DOMOVE        PLAYER'S MOVE, MUST BE TO AN ADJACENT ROOM
 *)
```

```
PROCEDURE DOMOVE;
VAR
        DEST : ROOM;

BEGIN
        READ(DEST);
        IF NOT (DEST IN [1..MAXROOMS]) THEN
                WRITELN('THERE IS NO ROOM #', DEST)
        ELSE IF NOT (DEST IN CAVE[PLAYER]) THEN
                WRITELN('I SEE NO TUNNEL TO ROOM #', DEST)
        ELSE
                 PLAYER := DEST;
        CHECKBATS;
        CHECKWUMP;
        CHECKPITS;
END; (* DOMOVE *)

(*
 * DOQUIT       ASKS IF THE PLAYER MEANT TO TYPE
 *              THE QUIT COMMAND.
 *)

PROCEDURE DOQUIT;
VAR
        ANSWER : CHAR;
BEGIN
        WRITELN;
        WRITE('DO YOU REALLY WANT TO QUIT NOW ? ');
        READ(ANSWER);
        QUITTING := ANSWER = 'Y';
END;


PROCEDURE DOATURN(ACTION : CHAR);
BEGIN
        CASE ACTION OF
        MOVE:
                DOMOVE;
        SHOOT:
                DOSHOOT;
        QUIT:
                DOQUIT;
        HELP:
                DOINSTR;
        END; (* CASE *)
        READLN;

END; (* DOATURN *)
```

```
(*
 * GAMEOVER      RETURNS TRUE IF THE GAME IS OVER
 *
 *)

FUNCTION GAMEOVER:BOOLEAN;
BEGIN
        GAMEOVER := QUITTING OR KILLED OR
                    WUMPUSKILLED OR (ARROWSLEFT = 0);
END;

(*
 * INITIALIZE    GENERATES A RANDOM MAZE AND THE
 *               POSITIONS OF THE PLAYER, WUMPUS, AND BATS.
 *               MAKE SURE THAT THE PLAYER DOESN'T START
 *               WITH THE WUMPUS.
 *)

PROCEDURE INITIALIZE;
VAR
        I : ROOM;

BEGIN
        RANDOMIZE;
        FOR I := 1 TO MAXROOMS DO               (* START WITH EMPTY MAZE *)
                CAVE[I] := [];
        BATS := [];
        PITS := [];
        MAKEMAZE;
        WUMPUS := RAND(1, MAXROOMS);
        FOR I := 1 TO MAXBATS DO
                BATS := BATS + [RAND(1, MAXROOMS)];
        FOR I := 1 TO MAXPITS DO
                PITS := PITS + [RAND(1, MAXROOMS)];
        REPEAT
           PLAYER := RAND(1, MAXROOMS);
        UNTIL (PLAYER <> WUMPUS) AND
              NOT (PLAYER IN PITS) AND
              NOT (PLAYER IN BATS);
        QUITTING := FALSE;
        KILLED := FALSE;
        WUMPUSKILLED := FALSE;
        ARROWSLEFT := NUMBEROFARROWS;
        COMMANDSET:= [ MOVE, SHOOT, QUIT, HELP];
END; (* INITIALIZE *)

BEGIN           (* MAIN PROGRAM *)
        WRITELN('WELCOME TO WUMPUS!!');
        ASKINSTRUCT;
        INITIALIZE;
        REPEAT
                DOATURN(COMMAND);
        UNTIL GAMEOVER;
END.
```


DL

# LANDER

**The Game.** This game puts you in a spaceship that is travelling through some distant galaxy. You are quickly approaching an unknown planet, where you hope to manage a successful landing. Quick thinking is of the essence; as every second ticks by the decisions you make determine whether the spacecraft—with you in it—will land safely or crash.

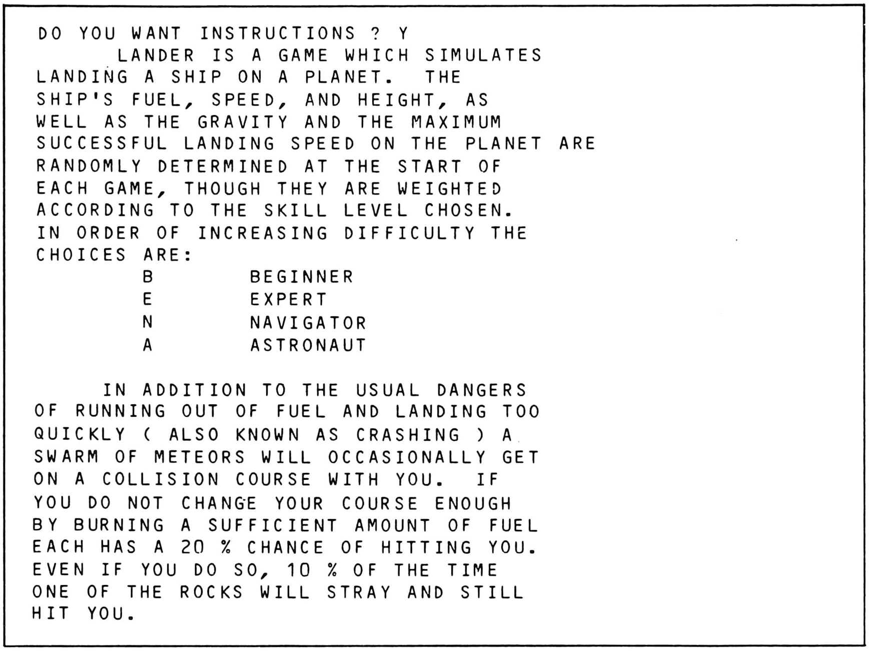
At the beginning of the game you are given the basic statistics of your situation: the velocity of your ship (in meters per second, toward the planet); your current distance from the planet ("height" in meters); and the amount of fuel you have left. Your job is to determine the right amount of fuel to burn to counteract the force of gravity that is hurling you toward the planet. You are given a second-by-second progress report. Unfortunately, you do not know the exact value of g (the acceleration due to gravity) for this planet (it ranges from about 9 to 11.5 m/sec$^2$ in this game). However, you do know that burning one unit of your fuel will accelerate your ship 1 m/sec$^2$ away from the planet. You maneuver your ship, then, by entering the quantity of fuel you wish to burn at each second of the landing operation. You respond to the prompt:

```
UNITS OF FUEL:
```

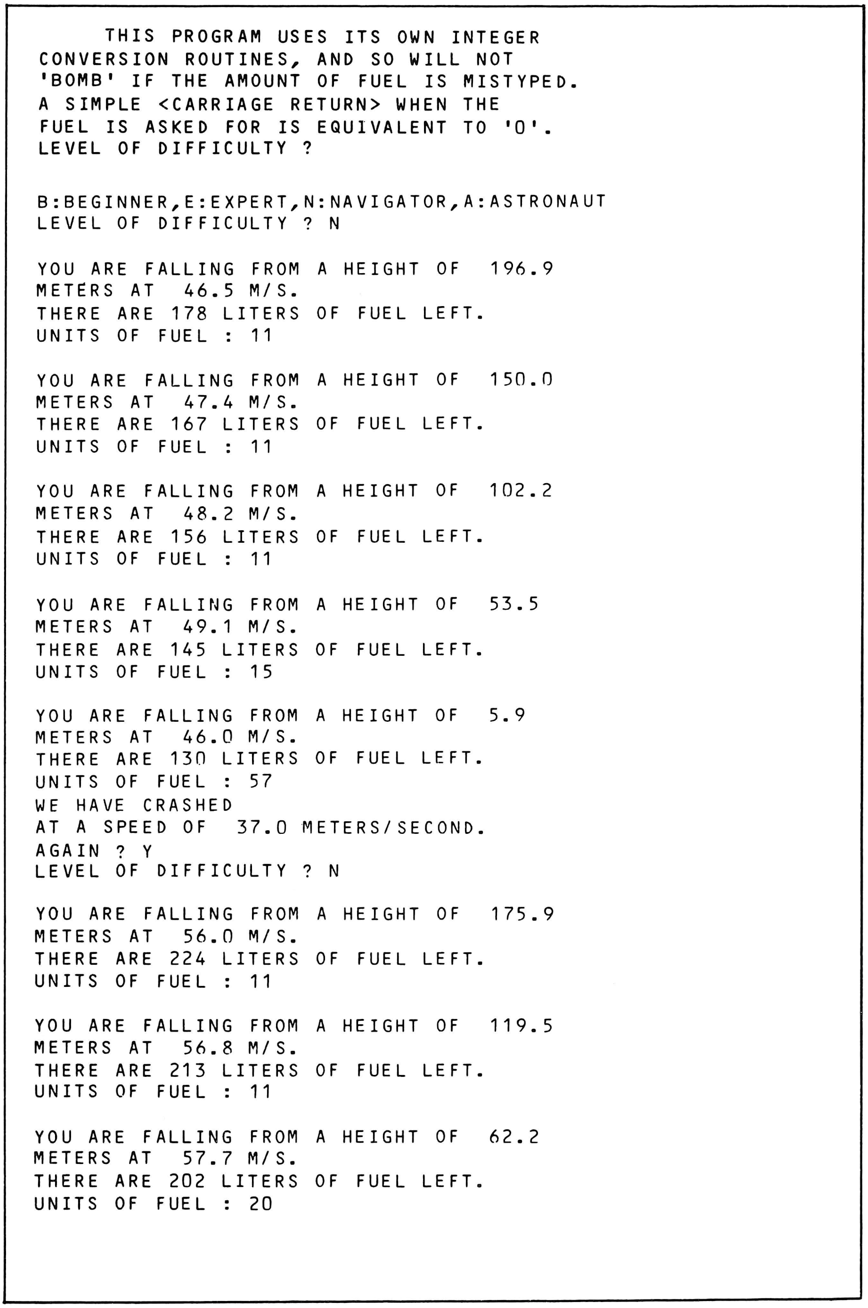
What makes your situation even more perilous is that there are many meteors circling this planet, and at any time you may find a cluster of them speeding toward you. If this happens, you may be forced to burn some of your precious fuel (at least 10 units are always required) to change your course and avoid collision with the meteors. If you burn the extra fuel, there is still a 10% chance that one of the meteors will hit you; if you do not, *each* meteor has a 20% chance of hitting you. (There may be from 2 to 12 meteors.)

Any one of three events will end this game: (1) a meteor hits you and destroys your ship; (2) you run out of fuel (or you exhibit inadequate skill in using the fuel you have) and you crash on the surface of the planet; or, the event to be hoped for, (3) you complete a successful "soft" landing on the planet.

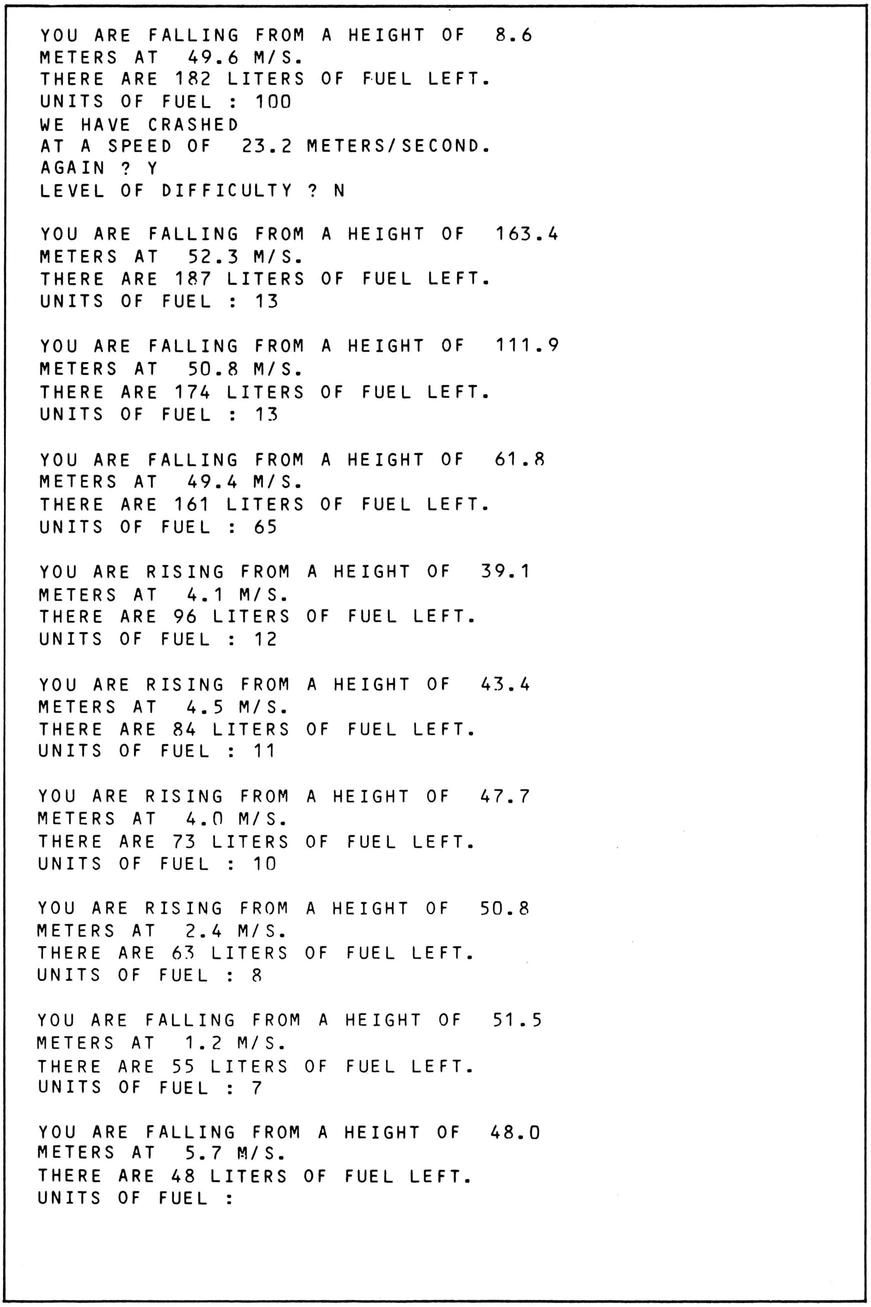
Notice that you are given the chance to choose the level of difficulty of this game: beginner, expert, navigator, or astronaut. (Enter a B, E, N, or A.) The level determines the *range* of all the statistics of your flight, including your velocity and distance from the planet, the amount of fuel you have, the gravity of the planet itself, and your likelihood of being hit by a meteor. The level also directly affects the maximum landing velocity that can be considered "soft" (ranging from 5 m/sec for a beginner down to 1 m/sec for an astronaut).



```
DO YOU WANT INSTRUCTIONS ? Y
     LANDER IS A GAME WHICH SIMULATES
LANDING A SHIP ON A PLANET.  THE
SHIP'S FUEL, SPEED, AND HEIGHT, AS
WELL AS THE GRAVITY AND THE MAXIMUM
SUCCESSFUL LANDING SPEED ON THE PLANET ARE
RANDOMLY DETERMINED AT THE START OF
EACH GAME, THOUGH THEY ARE WEIGHTED
ACCORDING TO THE SKILL LEVEL CHOSEN.
IN ORDER OF INCREASING DIFFICULTY THE
CHOICES ARE:
        B       BEGINNER
        E       EXPERT
        N       NAVIGATOR
        A       ASTRONAUT

     IN ADDITION TO THE USUAL DANGERS
OF RUNNING OUT OF FUEL AND LANDING TOO
QUICKLY ( ALSO KNOWN AS CRASHING ) A
SWARM OF METEORS WILL OCCASIONALLY GET
ON A COLLISION COURSE WITH YOU.  IF
YOU DO NOT CHANGE YOUR COURSE ENOUGH
BY BURNING A SUFFICIENT AMOUNT OF FUEL
EACH HAS A 20 % CHANCE OF HITTING YOU.
EVEN IF YOU DO SO, 10 % OF THE TIME
ONE OF THE ROCKS WILL STRAY AND STILL
HIT YOU.
```



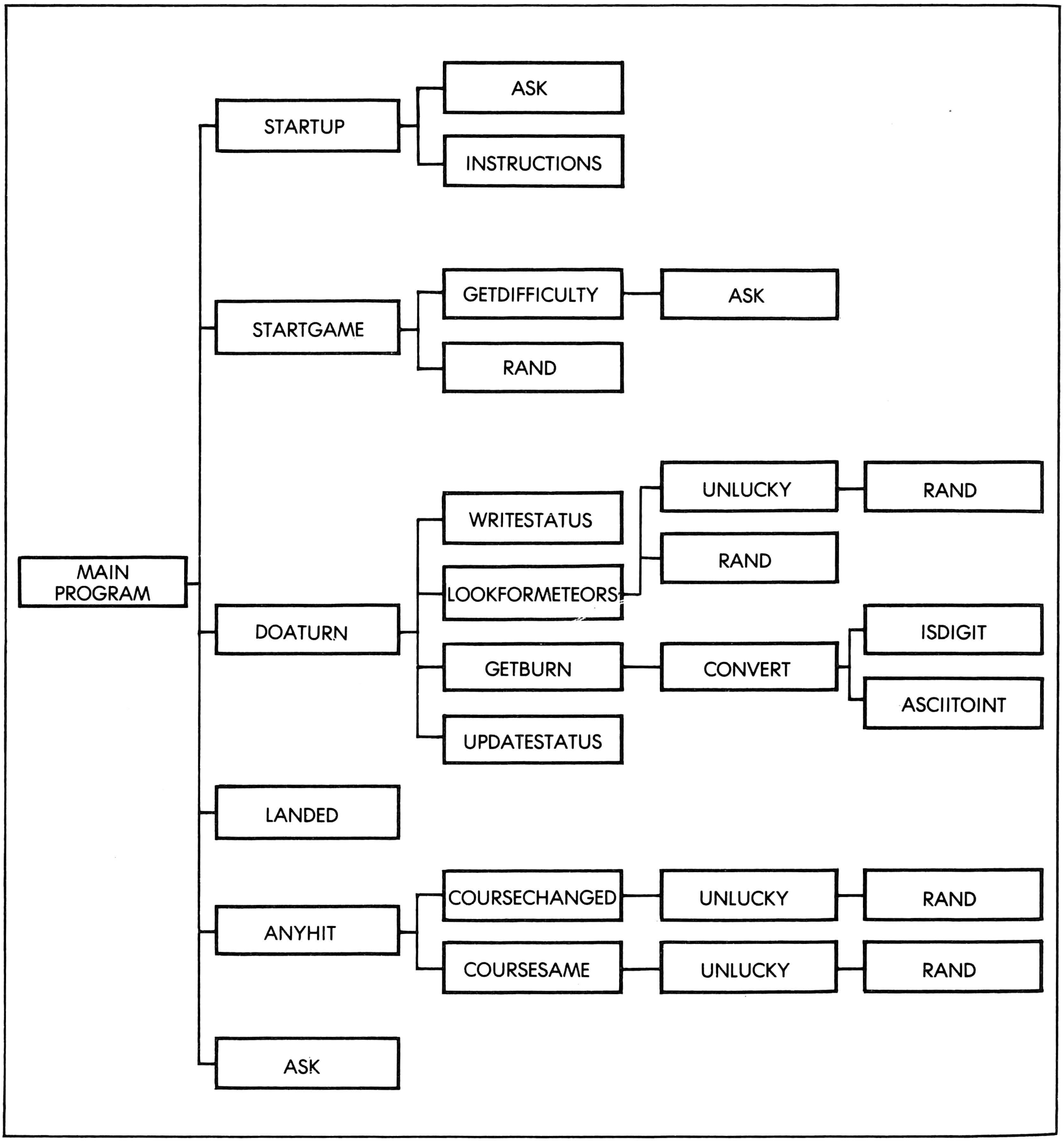
```
     THIS PROGRAM USES ITS OWN INTEGER
CONVERSION ROUTINES, AND SO WILL NOT
'BOMB' IF THE AMOUNT OF FUEL IS MISTYPED.
A SIMPLE <CARRIAGE RETURN> WHEN THE
FUEL IS ASKED FOR IS EQUIVALENT TO '0'.
LEVEL OF DIFFICULTY ?


B:BEGINNER,E:EXPERT,N:NAVIGATOR,A:ASTRONAUT
LEVEL OF DIFFICULTY ? N

YOU ARE FALLING FROM A HEIGHT OF  196.9
METERS AT  46.5 M/S.
THERE ARE 178 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE FALLING FROM A HEIGHT OF  150.0
METERS AT  47.4 M/S.
THERE ARE 167 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE FALLING FROM A HEIGHT OF  102.2
METERS AT  48.2 M/S.
THERE ARE 156 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE FALLING FROM A HEIGHT OF  53.5
METERS AT  49.1 M/S.
THERE ARE 145 LITERS OF FUEL LEFT.
UNITS OF FUEL : 15

YOU ARE FALLING FROM A HEIGHT OF  5.9
METERS AT  46.0 M/S.
THERE ARE 130 LITERS OF FUEL LEFT.
UNITS OF FUEL : 57
WE HAVE CRASHED
AT A SPEED OF  37.0 METERS/SECOND.
AGAIN ? Y
LEVEL OF DIFFICULTY ? N

YOU ARE FALLING FROM A HEIGHT OF  175.9
METERS AT  56.0 M/S.
THERE ARE 224 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE FALLING FROM A HEIGHT OF  119.5
METERS AT  56.8 M/S.
THERE ARE 213 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE FALLING FROM A HEIGHT OF  62.2
METERS AT  57.7 M/S.
THERE ARE 202 LITERS OF FUEL LEFT.
UNITS OF FUEL : 20
```



```
YOU ARE FALLING FROM A HEIGHT OF  8.6
METERS AT  49.6 M/S.
THERE ARE 182 LITERS OF FUEL LEFT.
UNITS OF FUEL : 100
WE HAVE CRASHED
AT A SPEED OF  23.2 METERS/SECOND.
AGAIN ? Y
LEVEL OF DIFFICULTY ? N

YOU ARE FALLING FROM A HEIGHT OF  163.4
METERS AT  52.3 M/S.
THERE ARE 187 LITERS OF FUEL LEFT.
UNITS OF FUEL : 13

YOU ARE FALLING FROM A HEIGHT OF  111.9
METERS AT  50.8 M/S.
THERE ARE 174 LITERS OF FUEL LEFT.
UNITS OF FUEL : 13

YOU ARE FALLING FROM A HEIGHT OF  61.8
METERS AT  49.4 M/S.
THERE ARE 161 LITERS OF FUEL LEFT.
UNITS OF FUEL : 65

YOU ARE RISING FROM A HEIGHT OF  39.1
METERS AT  4.1 M/S.
THERE ARE 96 LITERS OF FUEL LEFT.
UNITS OF FUEL : 12

YOU ARE RISING FROM A HEIGHT OF  43.4
METERS AT  4.5 M/S.
THERE ARE 84 LITERS OF FUEL LEFT.
UNITS OF FUEL : 11

YOU ARE RISING FROM A HEIGHT OF  47.7
METERS AT  4.0 M/S.
THERE ARE 73 LITERS OF FUEL LEFT.
UNITS OF FUEL : 10

YOU ARE RISING FROM A HEIGHT OF  50.8
METERS AT  2.4 M/S.
THERE ARE 63 LITERS OF FUEL LEFT.
UNITS OF FUEL : 8

YOU ARE FALLING FROM A HEIGHT OF  51.5
METERS AT  1.2 M/S.
THERE ARE 55 LITERS OF FUEL LEFT.
UNITS OF FUEL : 7

YOU ARE FALLING FROM A HEIGHT OF  48.0
METERS AT  5.7 M/S.
THERE ARE 48 LITERS OF FUEL LEFT.
UNITS OF FUEL :
```

```
YOU ARE FALLING FROM A HEIGHT OF  36.6
METERS AT  17.3 M/S.
THERE ARE 48 LITERS OF FUEL LEFT.
UNITS OF FUEL : 13

YOU ARE FALLING FROM A HEIGHT OF  20.0
METERS AT  15.8 M/S.
THERE ARE 35 LITERS OF FUEL LEFT.
UNITS OF FUEL : 28

YOU ARE RISING FROM A HEIGHT OF  12.4
METERS AT  0.6 M/S.
THERE ARE 7 LITERS OF FUEL LEFT.
UNITS OF FUEL : 1

YOU ARE FALLING FROM A HEIGHT OF  7.8
METERS AT  9.9 M/S.
THERE ARE 6 LITERS OF FUEL LEFT.
UNITS OF FUEL : 6
WE HAVE CRASHED
AT A SPEED OF  13.8 METERS/SECOND.
AGAIN ? N
BYE!
```

# The Program.


MAIN PROGRAM
STARTUP
ASK
INSTRUCTIONS
STARTGAME
GETDIFFICULTY
ASK
RAND
DOATURN
WRITESTATUS
LOOKFORMETEORS
UNLUCKY
RAND
RAND
GETBURN
CONVERT
ISDIGIT
ASCIITOINT
UPDATESTATUS
LANDED
ANYHIT
COURSECHANGED
UNLUCKY
RAND
COURSESAME
UNLUCKY
RAND
ASK

As we will see, LANDER illustrates several interesting programming techniques, including:

- A method of converting an input string into its corresponding numerical value (function CONVERT); this technique prevents the program from terminating prematurely just because "illegal" data are entered in response to an input request.
- A set of algorithms (in procedure STARTGAME and elsewhere) that incorporate the "level of difficulty" factor into every event of the game.
- A routine (function ASK) that prompts the user for input, validates the input, and supplies a second prompt if necessary; the versatile design of this routine permits its use in several different contexts.

The main program calls procedure STARTUP to offer instructions for the game. STARTUP makes the first call to function ASK. Notice the parameters of the call:

```
IF ASK('DO YOU WANT INSTRUCTIONS? ',
    'PLEASE ANSWER Y OR N ',
    ['Y','N']) = 'Y' THEN
    INSTRUCTIONS;
```

ASK displays the first prompt and reads the input. If the input character is **IN** the set (passed to the formal parameter RESPONSES), then ASK returns the character. Otherwise ASK displays the second prompt (IFBAD) and reads the new input:

```
IF CH IN RESPONSES THEN
  BAD := FALSE
ELSE
  WRITELN(IFBAD);
```

Procedure INSTRUCTIONS reads an external file (that contains instructions for the game) and writes each line of the file. The external file is opened with a RESET statement:

```
RESET(INSTRUCT,INSTRFILE);
```

INSTRFILE is a string constant, defined in the program heading:

```
INSTRFILE = 'INSTR.TEXT';
```

INSTR.TEXT is, of course, the name of the external file. INSTRUCT is defined as a file of TEXT in the **VAR** declaration section of procedure INSTRUCTIONS.

The UCSD function IORESULT returns a 0 for a normal, error-free input/output operation:

```
IF IORESULT <> 0 THEN
   WRITELN('INSTRUCTIONS ARE NOT AVAILABLE AT THIS TIME')
```

Until EOF(INSTRUCT) becomes TRUE, the file is read, line by line, into the string variable STR, and each value of STR is written:

```
WHILE NOT EOF(INSTRUCT) DO
BEGIN
   READLN(INSTRUCT,STR);
   WRITELN(STR);
END;
```

Next in the main program we see a pair of nested **REPEAT/UNTIL** loops. The outer loop initiates new games until ASK returns an N. The inner loop controls the rounds of a game by repeated calls to DOATURN until either function LANDED or function ANYHIT returns a TRUE.

The outer loop calls procedure STARTGAME to initialize the major variables of the game. STARTGAME begins with a call to procedure GETDIFFICULTY, which in turn calls function ASK to prompt the user for the level of difficulty of the game. GETDIFFICULTY then uses a **CASE** statement to assign the correct value to the variable CHANCE. (Notice that SLIGHTLY, MODERATELY, VERY, and IMPOSSIBLE are defined in the **CONST** section of the main program.)

Procedure STARTGAME then uses the value of CHANCE to define the *range* of four different variables (GRAVITY, HEIGHT, SPEED, and FUEL). Function RAND is called to pick a random value within the specified range. (Take a look at RAND; its algorithm is not the standard (RANDOM **MOD** MAX) + MIN that has been used elsewhere in this book. Instead, it uses the constant MAXRAND, which is set to the maximum value that RANDOM returns.)

STARTGAME also initializes MAXLANDINGSPEED to a value from 1 to 5, depending on the value of CHANCE. (CRASHLANDINGSPEED is set to 6.0 in the CONST section.):

```
MAXLANDINGSPEED := CRASHLANDINGSPEED — 10.0 * CHANCE;
```

The inner loop of the main program then begins its repeated calls to procedure DOATURN, which controls the second-by-second action of the spacecraft. DOATURN begins with a call to procedure WRITESTATUS (to display the statistics of the flight—i.e., the values of the variables that were set in STARTGAME) and then calls the somewhat more complicated procedure LOOKFORMETEORS.

For every round of the game, LOOKFORMETEORS determines whether or not the rocket is to encounter meteors. One function call at the beginning of the procedure determines whether the main algorithm of LOOKFORMETEORS will be executed:

```
IF UNLUCKY(0.1) THEN
```

UNLUCKY is a clever little function that performs a weighted flip of a coin. The probability of a TRUE is proportional to the value of the global variable CHANCE and the formal parameter PERCENTCHANCE:

```
UNLUCKY := RAND(0.0, 1.0 — CHANCE) <= PERCENTCHANCE;
```

If UNLUCKY returns a TRUE, then the main task of LOOKFORMETEORS is to determine the number of meteors that are racing toward the spacecraft, and the probability (MISSCHANCE) of all of the meteors *missing* the craft:

```
NUMMETEORS := ROUND(RAND(MINMETEORS,MAXMETEORS
                  + 10.0 * CHANCE));

FOR EACHROCK := 1 TO NUMMETEORS DO
  MISSCHANCE := MISSCHANCE * EACHMISSCHANCE;
```

Once again, MINMETEORS, MAXMETEORS, and EACHMISSCHANCE are all constants. Notice that the range of possible values of NUMMETEORS increases with the value of CHANCE. In other words, for the more advanced levels of the game there will probably be more intimidating groups of meteors heading for the spacecraft. Also, MISSCHANCE (which was initialized at 1.0) *decreases* in value each time the **FOR** loop executes its statement. For example, since EACHMISSCHANCE is equal to 0.8, a barrage of seven meteors have a 21% probability of missing the rocket (or a 79% probability of hitting it!). Furthermore, with twelve meteors (the maximum number in the most difficult level of the game) there is a staggering 93% probability that the rocket will be hit. We will see how the variable MISSCHANCE is used when we examine functions ANYHIT and COURSESAME.

LOOKFORMETEORS displays the bad news to the player. Next, DOATURN calls procedure GETBURN to receive the player's input for the quantity of fuel that is to be burned. GETBURN uses function CONVERT to interpret the input string.

The first **WHILE** loop of function CONVERT skips by any leading blanks in the input string. The second **WHILE** loop sends each nonblank character to function ISDIGIT to see if the character is a numerical digit, and then uses function ASCIITOINT and successive multiplications by 10 to convert the string into a numerical value:

```
INTVALUE := INTVALUE * 10 +
ASCIITOINT(NUMSTRING[POSITION]);
```

(ASCIITOINT uses the ORD function to convert a character to an integer.)

Once the user inputs a valid value for BURN, then DOATURN calls procedure UPDATESTATUS to calculate the resulting changes in the height and velocity of the spacecraft.

UPDATESTATUS performs its calculations in increments of one tenth of a second (TIMEINC) and continues through an entire second (DELTAT = TRUNC(1/TIMEINC)) of action. Two statements calculate the new height and the new velocity for every tenth of a second:

```
HEIGHT := HEIGHT - SPEED * TIMEINC -
              (GRAVITY - BURN) * 0.5 * SQR(TIMEINC);
SPEED := SPEED + (GRAVITY - BURN) * TIMEINC;
```

These statements correspond to the free-fall equations:

$$d_t = d_{t-1} - (vt + \tfrac{1}{2}gt^2)$$

and

$$v_t = v_{t-1} + gt$$

However, in the program lines the value of GRAVITY is decreased by the thrust of the burned fuel (BURN), which acts against the force of gravity.

After each call to DOATURN, the main program calls functions LANDED and ANYHIT to analyze the results of the previous second's action:

```
REPEAT
   DOATURN;
UNTIL LANDED OR ANYHIT;
```

LANDED returns a TRUE, and displays the appropriate message, when the height of the spacecraft is less than the constant LANDINGHEIGHT (0.1 meter).

ANYHIT returns a FALSE if there have been no meteors:

```
MISSCHANCE = CERTAINTY
```

or if the craft has already landed:

```
HEIGHT <= LANDINGHEIGHT
```

Otherwise ANYHIT calls either function COURSESAME or function COURSECHANGED, depending on the value of BURN. If the player burned over 10 units of fuel (BURN > MINAVOID), then an effort has been made to accelerate the craft out of the path of the swarm of meteors. COURSECHANGED is called, and in turn makes a rather lenient call to UNLUCKY to see if the player survived or not:

```
IF UNLUCKY(0.1) THEN
```

However, if the player has cavalierly ignored the computer's advice, and has burned less than the required amount of fuel, then the much more dangerous function COURSESAME is called. COURSESAME sends UNLUCKY the probability of a meteor hitting the ship:

```
UNLUCKY(1.0-MISSCHANCE)
```

Depending on whether UNLUCKY returns a TRUE or a FALSE, the good news or (much more likely) bad news is displayed to the player.

```
(*$I-*)                 (* TURN OFF I/O CHECKING, TO RECOVER FROM A *)
                        (* NON-EXISTENT INSTRUCTION FILE           *)
PROGRAM LANDER;

USES
   APPLESTUFF;          (* FOR RANDOM NUMBER FUNCTION *)
(*
 * THIS PROGRAM SIMULATES LANDING A SPACECRAFT.
 *)

CONST
   INSTRFILE     = 'INSTR.TEXT';

   MAXRAND       = 32767;(* LARGEST NUMBER THAT RANDOM RETURNS *)
   TIMEINC       = 0.1;  (* EXAMINE HOW THE SHIP IS MOVING *)
                        (* EVERY TIMEINC'TH OF A SECOND   *)
   SLIGHTLY      = 0.1;  (* THESE ARE THE DIFFICULTY   *)
   MODERATELY    = 0.2;  (* LEVELS OF THE GAME. THEY   *)
   VERY          = 0.3;  (* PARAMETERIZE HOW DIFFICULT *)
   IMPOSSIBLE    = 0.5;  (* IT WILL BE TO LAND SAFELY  *)

   CERTAINTY     = 1.0;

   MINMETEORS    = 2.0;  (* BOUNDS FOR THE NUMBER OF METEORS *)
   MAXMETEORS    = 7.0;  (* THAT MAY APPEAR IN A SHOWER      *)
   EACHMISSCHANCE = 0.8;(* CHANCE OF A SINGLE METEOR MISSING *)
   MINAVOID      = 10;   (* MINIMUM FUEL NEEDED FOR A GOOD   *)
                        (* CHANCE TO AVOID A SWARM          *)

   LANDINGHEIGHT = 0.1; (* HEIGHT AT WHICH WE SAY IT HAS LANDED *)
   CRASHLANDINGSPEED = 6.0;

TYPE
   ANSWERS = SET OF 'A' .. 'Z';

VAR
   SPEED, HEIGHT, GRAVITY : REAL;
   MAXLANDINGSPEED : REAL;
   CHANCE, MISSCHANCE : REAL;
   BURN, FUEL : INTEGER;
   NUMMETEORS : INTEGER;
   INP : STRING;                (* INPUT LINE FOR NUMBERS *)
   RANOUT : BOOLEAN;

(*
 * RETURNS A RANDOM REAL NUMBER BETWEEN LOWLIMIT AND
 * HIGHLIMIT.
 *)

FUNCTION RAND(LOWLIMIT, HIGHLIMIT : REAL) : REAL;
BEGIN
   RAND := LOWLIMIT + (HIGHLIMIT - LOWLIMIT + 1) * (RANDOM / MAXRAND);
END;
```

```
(*
 * RETURNS TRUE IF A RANDOM NUMBER, WEIGHTED
 * BY THE DIFFICULTY LEVEL, IS LESS THAN THE
 * ARGUMENT
 *)

FUNCTION UNLUCKY(PERCENTCHANCE : REAL) : BOOLEAN;
BEGIN
   UNLUCKY := RAND(0.0, 1.0 - CHANCE) <= PERCENTCHANCE;
END;

(*
 * CONVERTS A NUMERICAL ASCII CHARACTER TO
 * ITS CORRESPONDING INTEGRAL VALUE.
 *)

FUNCTION ASCIITOINT(DIGIT : CHAR) : INTEGER;
BEGIN
   ASCIITOINT := ORD(DIGIT) - ORD('0');
END;

(*
 * RETURNS TRUE IF THE ARGUMENT IS A NUMERICAL DIGIT.
 *)

FUNCTION ISDIGIT(CH : CHAR) : BOOLEAN;
BEGIN
   ISDIGIT := (CH >= '0') AND (CH <= '9');
END;

(*
 * CONVERT DETERMINES THE NUMERICAL VALUE OF
 * THE DIGITS IN A STRING.  IT RETURNS THE INTEGER
 * AND UPDATES THE STRING TO NOW CONTAIN WHATEVER
 * WAS AFTER THE END OF THE NUMBER.
 * IF THE STRING IS BLANK THE VALUE IS ZERO;
 * IF A NUMBER IS NOT FOUND IT RETURNS -1
 *)


FUNCTION CONVERT(NUMSTRING : STRING) : INTEGER;
VAR
   INTVALUE : INTEGER;        (* VALUE OF THE NUMBER *)
   POSITION : INTEGER;        (* CURRENT PLACE IN THE STRING *)
   DIGITSFOUND : INTEGER;     (* NUMBER OF NUMERICAL CHARS FOUND *)
   ENDOFNUMBER, NOTINTERESTING : BOOLEAN;
BEGIN
   INTVALUE := 0;
   DIGITSFOUND := 0;
   POSITION := 1;
   ENDOFNUMBER := FALSE;
   NOTINTERESTING := TRUE;
   IF LENGTH(NUMSTRING) > 0 THEN
   BEGIN
      WHILE (POSITION < LENGTH(NUMSTRING)) AND NOTINTERESTING DO
         IF NUMSTRING[POSITION] = ' ' THEN      (* SKIP BLANKS *)
            POSITION := POSITION + 1
         ELSE
            NOTINTERESTING := FALSE;
      WHILE (POSITION <= LENGTH(NUMSTRING)) AND (NOT ENDOFNUMBER) DO
         IF ISDIGIT(NUMSTRING[POSITION]) THEN
         BEGIN
            INTVALUE := INTVALUE * 10 + ASCIITOINT(NUMSTRING[POSITION]);
            POSITION := POSITION + 1;
            DIGITSFOUND := DIGITSFOUND + 1;
         END
         ELSE
            ENDOFNUMBER := TRUE;
      IF DIGITSFOUND = 0 THEN           (* RETURN -1 IF NO NUMBER SEEN *)
         INTVALUE := -1;
   END;
   CONVERT := INTVALUE;
END; (* CONVERT *)

(*
 * ASKS A QUESTION WITH A SINGLE CHARACTER ANSWER.  IF THE
 * RESPONSE IS IN THE SET 'ANSWERS' THE LETTER IS RETURNED.
 * OTHERWISE THE STRING IFBAD IS PRINTED AND THE QUESTION
 * IS ASKED AGAIN.
 *)

FUNCTION ASK(QUESTION, IFBAD : STRING; RESPONSES : ANSWERS) : CHAR;
VAR
   BAD : BOOLEAN;
   CH  : CHAR;
BEGIN
   BAD := TRUE;
   REPEAT
      WRITE(QUESTION);
      READ(CH);
      WRITELN;
      IF CH IN RESPONSES THEN
         BAD := FALSE
      ELSE
         WRITELN(IFBAD);
   UNTIL NOT BAD;                       (* WE MUST ASK AGAIN       *)
   ASK := CH;
END; (* ASK *)

(*
 * PRINTS OUT THE INSTRUCTIONS FROM A FILE INSTRFILE
 *)

PROCEDURE INSTRUCTIONS;
VAR
   INSTRUCT : TEXT;
   CH : CHAR;
   STR : STRING;
```

```
BEGIN
   RESET(INSTRUCT, INSTRFILE);
   IF IORESULT <> 0 THEN        (* NOT THERE *)
      WRITELN('INSTRUCTIONS ARE NOT AVAILABLE AT THIS TIME')
   ELSE                         (* FILE WAS FOUND, OPENED *)
      WHILE NOT EOF(INSTRUCT) DO
      BEGIN
          READLN(INSTRUCT, STR);
          WRITELN(STR);
      END;
END; (* INSTRUCTIONS *)

PROCEDURE STARTUP;
BEGIN
   IF ASK('DO YOU WANT INSTRUCTIONS ? ',
          'PLEASE ANSWER Y OR N',
          ['Y', 'N']) = 'Y' THEN
      INSTRUCTIONS;
   RANDOMIZE;
END; (* STARTUP *)

(*
 * ASK THE PLAYER FOR THE DIFFICULTY LEVEL OF THE
 * NEXT LANDING.
 *)

PROCEDURE GETDIFFICULTY;
VAR
   LEVEL : CHAR;
BEGIN
   LEVEL := ASK('LEVEL OF DIFFICULTY ? ',
                'B:BEGINNER,E:EXPERT,N:NAVIGATOR,A:ASTRONAUT',
                ['B', 'E', 'N', 'A']);
   CASE LEVEL OF
      'B' : CHANCE := SLIGHTLY:
      'E' : CHANCE := MODERATELY;
      'N' : CHANCE := VERY;
      'A' : CHANCE := IMPOSSIBLE;
   END;
END; (* GETDIFFICULTY *)

(*
 * VARIABLES THAT MUST BE RE-SET EACH TIME A
 * NEW LANDING IS ATTEMPED.
 *)

PROCEDURE STARTGAME;
BEGIN
   GETDIFFICULTY;
   GRAVITY := RAND(9.0 + CHANCE, 11.0 + CHANCE);
   HEIGHT  := RAND(1.0 + CHANCE, 2.0 + CHANCE) * 100.0;
   SPEED   := RAND(0.0, 100.0 * CHANCE) + 30.0;
   FUEL    := ROUND(50.0 * RAND(3.0 - CHANCE, 4.0 - CHANCE));
   MAXLANDINGSPEED := CRASHLANDINGSPEED - 10.0 * CHANCE;
   MISSCHANCE      := CERTAINTY;
   RANOUT  := FALSE;          (* JUST STARTING, HASN'T RUN OUT OF FUEL YET*)
END; (* STARTGAME *)
```

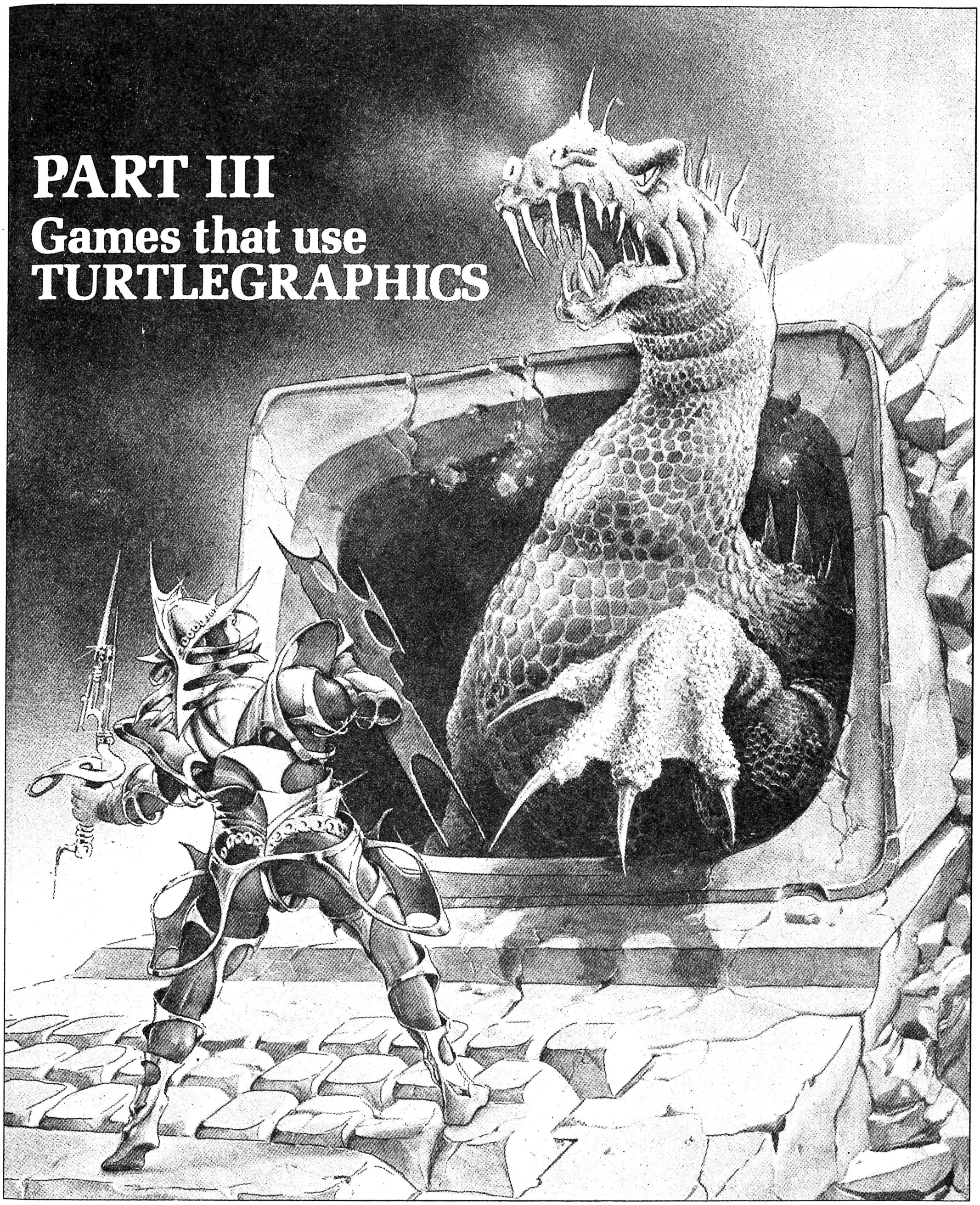
```
(*
 * TELL PLAYER HEIGHT, SPEED, AND DIRECTION.
 *)

PROCEDURE WRITESTATUS;
BEGIN
   WRITELN;
   WRITE('YOU ARE ');
   IF SPEED > 0.0 THEN
      WRITE('FALLING')
   ELSE
      WRITE('RISING');
   WRITELN(' FROM A HEIGHT OF ', HEIGHT:1:1);
   WRITELN('METERS AT ',ABS(SPEED):1:1, ' M/S.');
   IF NOT RANOUT THEN
      WRITELN('THERE ARE ', FUEL:1, ' LITERS OF FUEL LEFT.');
   RANOUT := FUEL = 0;
END; (* WRITESTATUS *)

(*
 * DETERMINES IF THERE ARE ANY METEORS, AND
 * IF SO, HOW MANY
 *)

PROCEDURE LOOKFORMETEORS;
VAR
   EACHROCK : INTEGER;
BEGIN
   MISSCHANCE := CERTAINTY;
   IF UNLUCKY( 0.1 ) THEN
   BEGIN
      NUMMETEORS := ROUND(RAND(MINMETEORS, MAXMETEORS + 10.0 * CHANCE));
      FOR EACHROCK := 1 TO NUMMETEORS DO
         MISSCHANCE := MISSCHANCE * EACHMISSCHANCE;
      WRITELN('WE ARE ON A COLLISION COURSE WITH ', NUMMETEORS:1);
      WRITE('METEORS.');
      IF RANOUT THEN            (* IF OUT OF FUEL DON'T TELL *)
         WRITELN
      ELSE
      BEGIN
         WRITELN('IF WE DO NOT USE MORE THAN ', MINAVOID:1);
         WRITELN('LITERS OF FUEL IN THE NEXT SECOND THERE');
         WRITELN('IS A ', ROUND(100.0 * (1.0 - MISSCHANCE)),
                 ' % PROBABILITY THAT WILL WE BE');
         WRITELN('HIT. IF MORE IS USED IT WILL BE ONLY 10 %.');
      END;
   END;
END; (* LOOKFORMETEORS *)

(*
 * ASKS THE PLAYER FOR THE AMOUNT OF FUEL
 * TO USE IN THE NEXT TIME PERIOD.
 *)
```

```
PROCEDURE GETBURN;
CONST
   FUELPROMPT = 'UNITS OF FUEL : ';
BEGIN
   REPEAT
      WRITE(FUELPROMPT);
      READLN(INP);
      BURN := CONVERT(INP);
      IF BURN > FUEL THEN
         WRITELN('THERE ISN'T  THAT MUCH FUEL LEFT.')
      ELSE IF BURN < 0 THEN
         WRITELN('I DON'T  THINK THAT''S POSSIBLE.');;
   UNTIL (BURN <= FUEL) AND (BURN >= 0);
END; (* GETBURN *)

(*
 * FIGURE OUT THE CRAFT'S NEW SPEED ACCORDING
 * TO THE LAWS OF PHYSICS.
 *)

PROCEDURE UPDATESTATUS;
VAR
   DELTAT : INTEGER;
BEGIN
   FUEL := FUEL - BURN;
   DELTAT := 0;
   REPEAT
      DELTAT := DELTAT + 1;
      HEIGHT := HEIGHT - SPEED * TIMEINC -
                 (GRAVITY - BURN) * 0.5 * SQR(TIMEINC);
      SPEED := SPEED + (GRAVITY - BURN) * TIMEINC;
   UNTIL (DELTAT = TRUNC(1 / TIMEINC)) OR (HEIGHT <= LANDINGHEIGHT);
END; (* UPDATESTATUS *)

(*
 * ALL THE PROCEDURES THAT MAKE A "TURN"
 *)
PROCEDURE DOATURN;
BEGIN
   WRITESTATUS;
   LOOKFORMETEORS;
   IF FUEL > 0 THEN     (* DON'T ASK IF NO MORE FUEL *)
      GETBURN
   ELSE
      BURN := 0;
   UPDATESTATUS;
END; (* DOATURN *)

(*
 * THE COURSE HAS CHANGED :
 * METEORS HAVE A LOWER CHANCE OF HITTING
 *)
```

```
FUNCTION COURSECHANGED : BOOLEAN;
BEGIN
   IF UNLUCKY(0.1) THEN
   BEGIN
      WRITELN('DESPITE THE PRECAUTIONARY MEASURES');
      WRITELN('TAKEN THE SHIP WAS DESTROYED.');
      COURSECHANGED := TRUE;
   END
   ELSE
   BEGIN
      WRITELN('YOUR PRUDENT ACTIONS HAVE SAVED THE');
      WRITELN('SHIP FROM THE MENACING METEORS!');
      COURSECHANGED := FALSE;
   END
END; (* COURSECHANGED *)

FUNCTION COURSESAME : BOOLEAN;
BEGIN
   IF NOT UNLUCKY(1.0 - MISSCHANCE) THEN
   BEGIN
      WRITELN('YOUR PILOTING SKILLS HAVE STEERED YOU');
      WRITELN('THROUGH THE CENTER OF THE SWARM!');
      COURSESAME := FALSE;
   END
   ELSE IF RANOUT THEN      (* DIDN'T HAVE THE FUEL TO BURN, NOT HIS FAULT *)
   BEGIN
      WRITELN('WHAT A PITY .. YOUR CRAFT WAS DEMOLISHED BY');
      WRITELN('METEORS BEFORE IT COULD BE VAPORIZED');
      WRITELN('ON CONTACT WITH THE SURFACE.');
      COURSESAME := TRUE;
   END
   ELSE
   BEGIN
      WRITELN('YOUR POINTLESS GAMBLING HAS DESTROYED');
      WRITELN('THE SHIP, YOU FOOLISH PLEBE!');
      COURSESAME := TRUE;
   END;
END; (* COURSESAME *)

(*
 * FIGURES IF ANY METEORS ( IF THERE WERE ANY ) MANAGED TO HIT THE SHIP.
 * DIFFERENT MESSAGES ARE PRINTED DEPENDING ON THE THRUST OF THE LAST
 * TURN.
 *)

FUNCTION ANYHIT : BOOLEAN;
BEGIN
   IF (MISSCHANCE = CERTAINTY) OR (HEIGHT <= LANDINGHEIGHT) THEN
      ANYHIT := FALSE             (* SKIP IF NO METEORS OR ALREADY ON GROUND *)
   ELSE IF BURN > MINAVOID THEN
      ANYHIT := COURSECHANGED
   ELSE                 (* DIDN'T BURN ENOUGH FUEL TO CHANGE COURSE *)
      ANYHIT := COURSESAME;
END; (* ANYHIT *)
```

```
(*
 * RETURNS TRUE IF THE SHIP HAS COME CLOSE ENOUGH TO
 * THE GROUND THAT WE CAN SAY THAT IT HAS LANDED.
 * COULD CRASH OR TOUCH DOWN SAFELY.
 *)

FUNCTION LANDED : BOOLEAN;
BEGIN
   IF HEIGHT < LANDINGHEIGHT THEN       (* CLOSE TO THE GROUND ? *)
   BEGIN
      SPEED := ABS(SPEED);
      IF SPEED < MAXLANDINGSPEED THEN
         WRITELN('WE HAVE LANDED SAFELY')
      ELSE
         WRITELN('WE HAVE CRASHED');
      WRITELN('AT A SPEED OF ', SPEED:1:1, ' METERS/SECOND.');
      LANDED := TRUE;
   END
   ELSE
      LANDED := FALSE;
END; (* LANDED *)

BEGIN          (* MAIN PROGRAM *)
   STARTUP;
   REPEAT
      STARTGAME;
      REPEAT
         DOATURN;
      UNTIL LANDED OR ANYHIT;
   UNTIL ASK('AGAIN ? ', 'Y OR N', ['Y', 'N']) = 'N';
   WRITELN('BYE!');
END.
```

# PART III
## Games that use TURTLEGRAPHICS

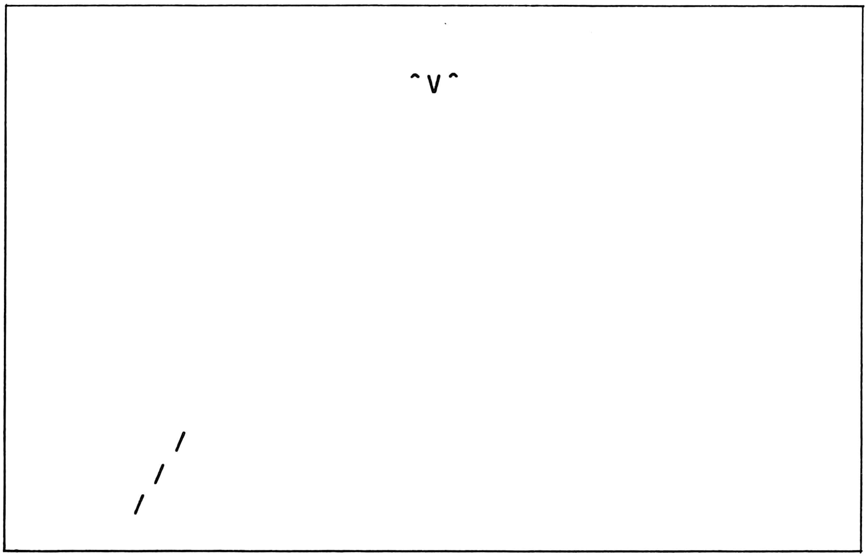


# GUNSHOT

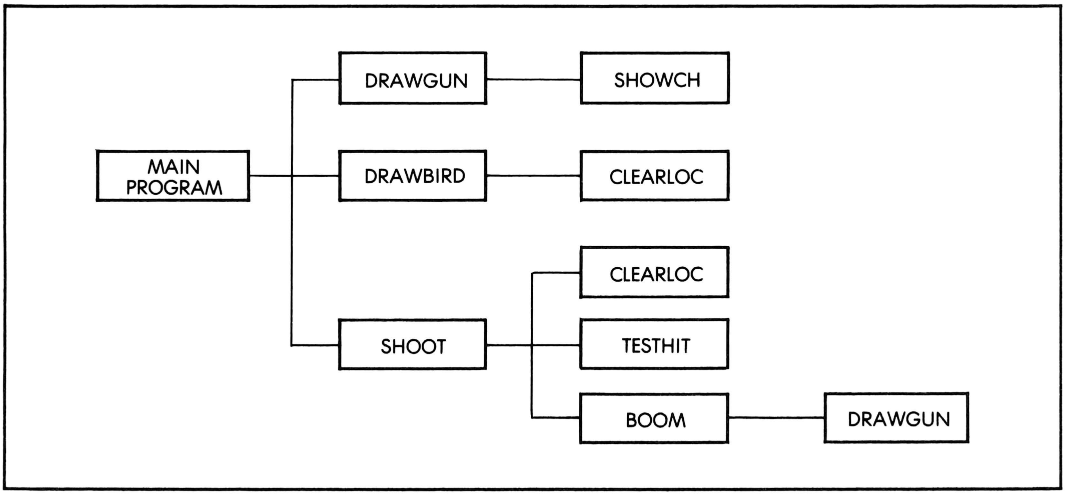
**The Game.** . . . The great northeastern swampland. Dawn. Silence except for the constant drizzle of cold rain. You are crouching in the mud, shivering, thinking wistfully of the comforts of the hearth, yet waiting patiently for the arrival of the big birds. Suddenly you hear a distant "Honk! honk!" and you know the moment has arrived. Soon the unsuspecting birds will be flying overhead and you will once again act out the minidrama of man vs. nature, predator vs. prey. Will the hunt be good today? Time will tell . . .

Your gun appears in the lower left-hand corner of the screen, and the birds fly across the top line. Press the RETURN key to fire the gun. A direct hit is indicated by a white rectangle in the bird's path and a moment of stopped action.

When you are ready to return home for some hot cocoa and flapjacks, press the Q to end the game.


^v^
/
/
/

# The Program.


MAIN PROGRAM
DRAWGUN
SHOWCH
DRAWBIRD
CLEARLOC
SHOOT
CLEARLOC
TESTHIT
BOOM
DRAWGUN

The screen in Turtlegraphics is 279 dots wide by 191 dots high. The important type declaration in this program is thus LOC, which is a record type with two fields:

```
LOC = RECORD
   XPOS : INTEGER;
   YPOS : INTEGER
END;
```

The fields will represent the Cartesian coordinates of any dot on the screen. Both BIRDLOC and SHOTLOC are variables of type LOC.

The main program begins by initializing BIRDLOC and SHOTLOC. (Remember that the lower left-hand corner of the screen has the coordinates (0,0) in Turtlegraphics.) The *y* coordinate of BIRDLOC is set at 179 (the constant BIRDY), because that is the row of dots where the "bird" characters will always be written. BIRDLOC.XPOS is initialized to zero. The initial shot location is (21,21), represented by the constant START:

```
SHOTLOC.XPOS := START;
SHOTLOC.YPOS := START;
```

As we will see, this point is the tip of the gun drawn in procedure DRAWGUN.

After initializing the BOOLEAN variable FIRE to FALSE, the program prompts the user to hit RETURN to start the game. A call to INITTURTLE puts the screen in graphics mode. Then DRAWGUN is called.

Procedure DRAWGUN and its subprocedure SHOWCH write three slash characters (/) in the lower left-hand corner of the screen to represent the gun. The main block of DRAWGUN consists of three calls to SHOWCH. The parameters of each call are the (x,y) coordinates of the bottom dot of the slash to be written.

Procedure SHOWCH contains three statements. First:

```
MOVETO(XPOS,YPOS);
```

This statement moves the turtle to the position (XPOS,YPOS). (Since INITTURTLE sets the pen color to NONE, the turtle's path does not appear on the screen for this MOVETO.) Next:

```
CHARTYPE(10);
```

This selects the character mode (10), allowing a white character to be printed on a black screen. Finally:

```
WCHAR('/');
```

WCHAR is the procedure that actually writes a character in graphics mode. The coordinates of the lower left corner of the character are determined by the turtle's current position (in this case, the coordinates passed to SHOWCH from DRAWGUN).

Next, the main program enters a **WHILE** loop that only ends if the player types a Q. This loop contains calls to the procedures that actually draw the birds and the shots. The key to the moving graphics illusion produced by this program is the speed with which this **WHILE** loop is continually executed.

Procedure DRAWBIRD is called at the beginning of the loop. DRAWBIRD writes one character each time it is called—either an up-arrow (^) or a V, representing the wings and the body of the bird, respectively. It begins by calling procedure CLEARLOC to remove the previous character that was displayed for the bird. CLEARLOC is sent the present (x,y) coordinates of the bird:

```
CLEARLOC(BIRDLOC.XPOS,BIRDY);
```

Procedure CLEARLOC begins by setting the pen color to NONE, and moving the turtle to the (x,y) position represented by the formal parameters XPOS and YPOS. Then the lines:

```
CHARTYPE(0);
WCHAR(' ')
```

effectively blank out the bird character, since the 0 mode of procedure CHARTYPE fills the indicated area of the screen with black.

The next routine in DRAWBIRD tests the location of the bird to see if it has reached the end (right side) of the line:

```
IF BIRDLOC.XPOS >= MAXX THEN . . .
```

If it has, then the x coordinates of both BIRDLOC and the turtle are moved back to zero:

```
BIRDLOC.XPOS := 0;
MOVETO(0,BIRDY)
```

An **IF/THEN/ELSE** statement then sets LASTBIRD to the *alternate* bird character (i.e., ^ or V).

The next statement may seem slightly confusing:

```
BIRDLOC.XPOS := TURTLEX;
```

The function TURTLEX returns the present x position of the turtle. Whenever a character is written (through execution of the WCHAR statement), the x coordinate of the turtle is advanced 7 dots. Thus the statement above "increments" BIRDLOC.XPOS after the WCHAR statement of procedure CLEARLOC.

Finally, the bird character is written:

```
CHARTYPE(10);
WCHAR(LASTBIRD);
```

In order to slow the action down a bit, an empty loop is executed 25 times (as long as the player hasn't just fired the gun):

```
IF NOT FIRE THEN
  FOR WAIT := 1 TO WAITMAX DO
    ;
```

The next lines in the main program may also seem a bit cryptic:

```
IF KEYPRESS THEN
  BEGIN
    READ(KEYBOARD,CH);
    . . .
```

The function KEYPRESS returns a TRUE value if a key has been pressed on the keyboard either since the beginning of the program or since the last READ statement. Thus, the statement above READs the most recently entered character. If the character is a blank, then FIRE is set to TRUE and procedure SHOOT is called.

Procedure SHOOT produces a white cursor-like square (representing the bullet) each time it is called. It begins by calling CLEARLOC to erase the previous bullet. Then the XPOS and YPOS fields of SHOTLOC are each incremented by CHARSIZE (a constant set at 7). The following three statements then move the turtle to the appropriate position and produce the new bullet:

```
MOVETO(SHOTLOC.XPOS,SHOTLOC.YPOS);
CHARTYPE(15);
WCHAR(' ');
```

(The 15 mode of CHARTYPE fills the indicated area of the screen with white.)

When the bullet reaches the line where the bird travels, FIRE is set to FALSE:

```
FIRE := NOT(SHOTLOC.YPOS >= BIRDY);
```

Then, in a single statement, function TESTHIT and procedure BOOM are called:

```
IF TESTHIT THEN
   BOOM;
```

Function TESTHIT returns a TRUE if the x and *y* components of SHOTLOC are within 3 (the constant WIDTH) dots from the x and *y* components of BIRDLOC. If TESTHIT returns a TRUE, then procedure BOOM produces a white "hit" pattern in the vicinity of the shot bird. (Note that CHARTYPE is set to mode 15.) BOOM also reinitializes FIRE (to FALSE) and BIRDLOC.XPOS (to 0). When the hit pattern is drawn, BOOM enters a waiting loop:

```
FOR WAIT := 1 TO 1000 DO
   ;
```

Finally, the FILLSCREEN procedure is used to erase everything on the screen, and DRAWGUN is called to draw the gun once again.

Back in procedure SHOOT, once FIRE has returned to a value of FALSE, the last bullet is erased (through a call to CLEARLOC), and the (*x*,*y*) coordinates of SHOTLOC are reinitialized to START.

At the end of the game, the main program returns to TEXTMODE (out of GRAFMODE, which was initiated by the INITTURTLE procedure).

```
(*
 * GUN
 * A STATIONARY GUN SHOOTS AT A MOVING BIRD
 *)
PROGRAM GUN(INPUT,OUTPUT);
USES TURTLEGRAHICS, APPLESTUFF;
CONST
   BIRDY   = 179;         (* THE LINE WHERE THE BIRD FLIES *)
   MAXX    = 279;         (* THE MAXIMUM SIZE IN THE X DIRECTION FOR SCREEN *)
   HITLOC  = 189;         (* WHERE THE SHOT CROSSES THE BIRD'S PATH *)
   WIDTH   =   3;         (* THE WIDTH FROM THE CENTER OF THE SHOT *)
   CHARSIZE=   7;         (* THE SIZE OF A CHARACTER ON THE SCREEN *)
   START   =  21;         (* THE SHOT STARTS FROM HERE *)

TYPE
   LOC = RECORD
      XPOS : INTEGER;     (* THE X LOCATION ON THE SCREEN *)
      YPOS : INTEGER      (* THE Y LOCATION ON THE SCREEN *)
   END;

VAR
   BIRDLOC : LOC;         (* WHERE THE BIRD IS *)
   SHOTLOC : LOC;         (* WHERE THE SHOT IS *)
   CH      : CHAR;        (* THE CHARACTER ON THE KEYBOARD *)
   LASTBIRD: CHAR;        (* THE LAST CHARACTER DRAWN FOR THE BIRD *)
   FIRE    : BOOLEAN;     (* WHETHER WE ARE FIRING *)



   (*
    * DRAWGUN
    * DRAWS THE GUN
    *)
   PROCEDURE DRAWGUN;
      (*
       * SHOWCH
       * PRINTS A '/' AT THE GIVEN SPOT
       *)
      PROCEDURE SHOWCH(XPOS,YPOS:INTEGER);
      BEGIN
         MOVETO(XPOS,YPOS);
         CHARTYPE(10);
         WCHAR('/');
      END; (* SHOWCH *)
   BEGIN
      SHOWCH(0,0);
      SHOWCH(CHARSIZE,CHARSIZE);
      SHOWCH(CHARSIZE*2,CHARSIZE*2)
   END; (* DRAWGUN *)


   (*
    * CLEARLOC
    * PUTS A SPACE AT LOCATION
    * XPOS, AND YPOS
    *)
```

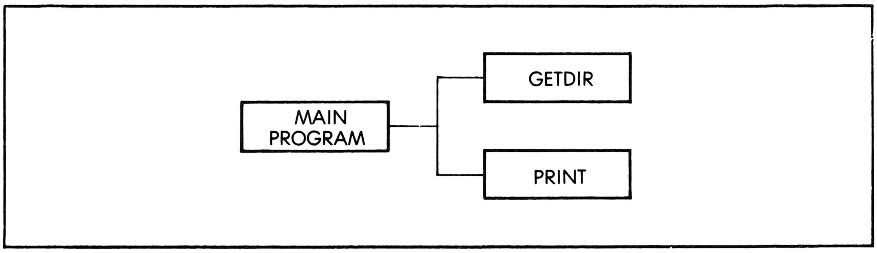
```
PROCEDURE CLEARLOC(XPOS,YPOS:INTEGER);
BEGIN

   PENCOLOR(NONE);
   IF XPOS >= 0 THEN
   BEGIN
      MOVETO(XPOS,YPOS);
      CHARTYPE(0);              (* WILL PUT A BLANK OUT *)
      WCHAR(' ')
   END
   ELSE
      MOVETO(0,0)
END;

(*
 * DRAWBIRD
 * REMOVES THE OLD BIRD, AND DRAWS A NEW ONE.
 *)

PROCEDURE DRAWBIRD;
CONST
   WAITMAX  = 25;              (* HOW LONG TO DO NOTHING *
VAR
   WAIT  :  1..WAITMAX;       (* USED TO SLOW BIRD DOWN *)
BEGIN

   CLEARLOC(BIRDLOC.XPOS,BIRDY);
   (*
    * WRAP AROUND THE SCREEN IF NEEDED
    *)
   IF BIRDLOC.XPOS >= MAXX THEN
   BEGIN
      BIRDLOC.XPOS := 0;
      MOVETO(0,BIRDY)
   END;

   (*
    * PUT OUT A
    * '^' OR A 'V'
    *)

   IF LASTBIRD = '^' THEN
      LASTBIRD := 'V'
   ELSE
      LASTBIRD := '^';


   BIRDLOC.XPOS := TURTLEX;


   CHARTYPE(10);              (* PLACE CHARACTER ON SCREEN *)
   WCHAR(LASTBIRD);
   (*
    * SLOW DOWN BIRD IF WE ARE NOT
    * SHOOTING
    *)
   IF NOT FIRE THEN
      FOR WAIT := 1 TO WAITMAX DO
         ;                           (* NOTHING *)
END; (* DRAWBIRD *)


(*
 * SHOOT
 * CLEARS THE CURRENT 'BULLET', AND DRAWS
 * ANOTHER ONE ON THE LINE ABOVE
 *)
PROCEDURE SHOOT;

   (*
    * TESTHIT
    * SEE IF WE HAVE HIT THE BIRD
    *)
   FUNCTION TESTHIT:BOOLEAN;
   VAR
      RETVAL  : BOOLEAN;             (* RETURN VALUE *)
   BEGIN
      RETVAL := FALSE;

      RETVAL := (SHOTLOC.YPOS <= (BIRDY +WIDTH)) AND
                (SHOTLOC.YPOS >= (BIRDY-WIDTH));
      TESTHIT := RETVAL AND
              ( (SHOTLOC.XPOS <= (BIRDLOC.XPOS+WIDTH))AND
                (SHOTLOC.XPOS >=(BIRDLOC.XPOS-WIDTH)))
   END; (* TESTHIT *)

   (*
    * BOOM
    * WE HIT THE THING
    *)
   PROCEDURE BOOM;
   VAR
      WAIT : 1..1000;                (* KEEP A STEADY PICTURE *)
   BEGIN
      FIRE := FALSE;
      BIRDLOC.XPOS := 0;

      TURNTO(180);
      MOVE(7);
      CHARTYPE(15);
      WSTRING('   ');
      TURNTO(90);
      MOVE(7);
      TURNTO(180);
      MOVE(17);
      WSTRING('  ');
      TURNTO(90);
      MOVE(7);
```

```
         TURNTO(180);
         MOVE(10);
         WCHAR(' ');
         FOR WAIT := 1 TO 1000 DO
            ;            (* NOTHING *)
         FILLSCREEN(BLACK);
         DRAWGUN
      END; (* BOOM *)
   BEGIN
      CLEARLOC(SHOTLOC.XPOS,SHOTLOC.YPOS );

      (*
       * GET NEW LOCATIONS
       *)
      SHOTLOC.XPOS := SHOTLOC.XPOS + CHARSIZE;
      SHOTLOC.YPOS := SHOTLOC.YPOS + CHARSIZE;
      MOVETO(SHOTLOC.XPOS,SHOTLOC.YPOS);
      CHARTYPE(15);              (* MAKES A SQUARE *)
      WCHAR(' ');
      FIRE := NOT (SHOTLOC.YPOS >= BIRDY);

      IF TESTHIT THEN
         BOOM;
      IF NOT FIRE THEN
      BEGIN
         CLEARLOC(SHOTLOC.XPOS,SHOTLOC.YPOS);
         SHOTLOC.XPOS := START;
         SHOTLOC.YPOS := START
      END
   END; (* SHOOT *)
BEGIN
   BIRDLOC.YPOS := BIRDY;
   BIRDLOC.XPOS := 0;
   SHOTLOC.XPOS := START;
   SHOTLOC.YPOS := START;
   FIRE := FALSE;
   WRITE('WHEN READY, HIT <RETURN> ');
   READLN(INPUT);
   INITTURTLE;
   DRAWGUN;
   WHILE CH <> 'Q' DO
   BEGIN
      DRAWBIRD;
      IF KEYPRESS THEN
      BEGIN
         READ(KEYBOARD,CH);
         IF CH = ' ' THEN
            FIRE := TRUE
      END;
      IF FIRE THEN
         SHOOT
   END;

   TEXTMODE
END.
```

# PICTURE

**The Game.** You can use this program to draw pictures on the screen with a "paddle" input device. The cursor leaves a display of its path as it moves around the screen. You can change the direction of the cursor by turning the knob of the paddle; the new direction will be indicated by a character (^, V, <, or >) displayed at the bottom left corner of the screen. Simply press the button to move the cursor. Enter an R from the keyboard to produce a "negative" of the image on the screen. To erase the current drawing, press a C (for a dark screen) or a W (for a white screen).

## The Program.



This program uses many of the same Turtlegraphics procedures that are used in the Gunshot program.

The main program continually calls GETDIR to determine the direction of the cursor, and PRINT to move the cursor and display the "path". (The procedure calls are inside a **WHILE** loop.) The BUTTON function returns a TRUE if the control button is pressed and a FALSE if it is not:

```
IF BUTTON(0) THEN
   PRINT;
```

Procedure GETDIR receives the direction input from the paddle (via a call to function PADDLE) and stores the value (an integer from 1 to 255) in variable PADANG:

```
PADANG := PADDLE(0);
```

This input is converted into one of four directions (DOWN, LEFT, UP, RIGHT). The turtle is then moved to the lower left corner of the screen:

```
MOVETO(0,0)
```

and a **CASE** statement is used to select the appropriate direction character to write there.

The last statement of GETDIR moves the turtle back to its original position:

```
MOVETO(XPOS,YPOS)
```

XPOS and YPOS were determined through calls to TURTLEX and TURTLEY before the turtle's first move. Recall that INITTURTLE places the turtle at the center of the screen, facing right.

Procedure PRINT uses the TURNTO and MOVE procedures to position the turtle for the next cursor display. For example:

```
TURNTO(LEFTANG);
MOVE(NEXTCH)
```

(Notice the definitions of LEFTANG, DOWNANG, UPANG, NEXTCH, and NEXTLINE in the **CONST** section.) No action is taken for RIGHT because the turtle is already in a position to write a character to the right; the turtle's position always defines the lower left dot of any character that is written in GRAFMODE.

Finally, the cursor is written:

```
CHARTYPE(3);
WCHAR(' ');
```

Mode 3 of CHARTYPE displays the *complement* of whatever is already on the screen. In this case, white will be changed to black, and black will be changed to white. Thus, if the cursor crosses its own path it will leave a black square at the point of intersection.

Unless the direction is RIGHT, the turtle must next be moved back to the position it held before the character was written. (Recall that a WCHAR statement moves the turtle 7 dots to the right.):

```
IF DIRECTION <> RIGHT THEN
  BEGIN
    TURNTO(LEFTANG);
    MOVE(NEXTCH)
  END;
```

Now the turtle is in position for its next move.

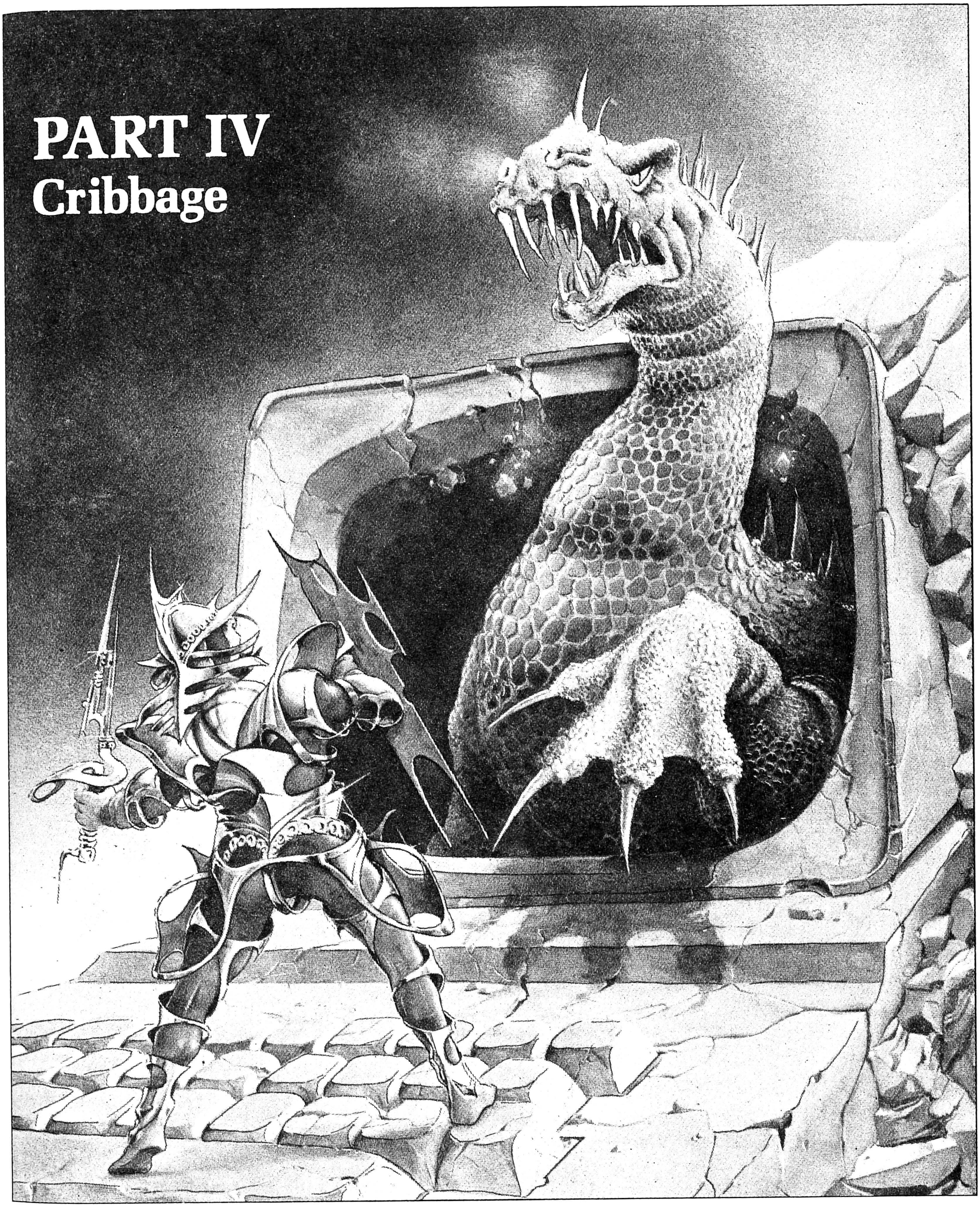
Meanwhile, back in the **WHILE** loop of the main program, the function KEYPRESS returns a TRUE if a character has been entered from the keyboard. The character is read, and the appropriate call to the FILLSCREEN procedure is made from a **CASE** statement. R reverses the image (to black on white); C blanks out the screen with black, and W with white.

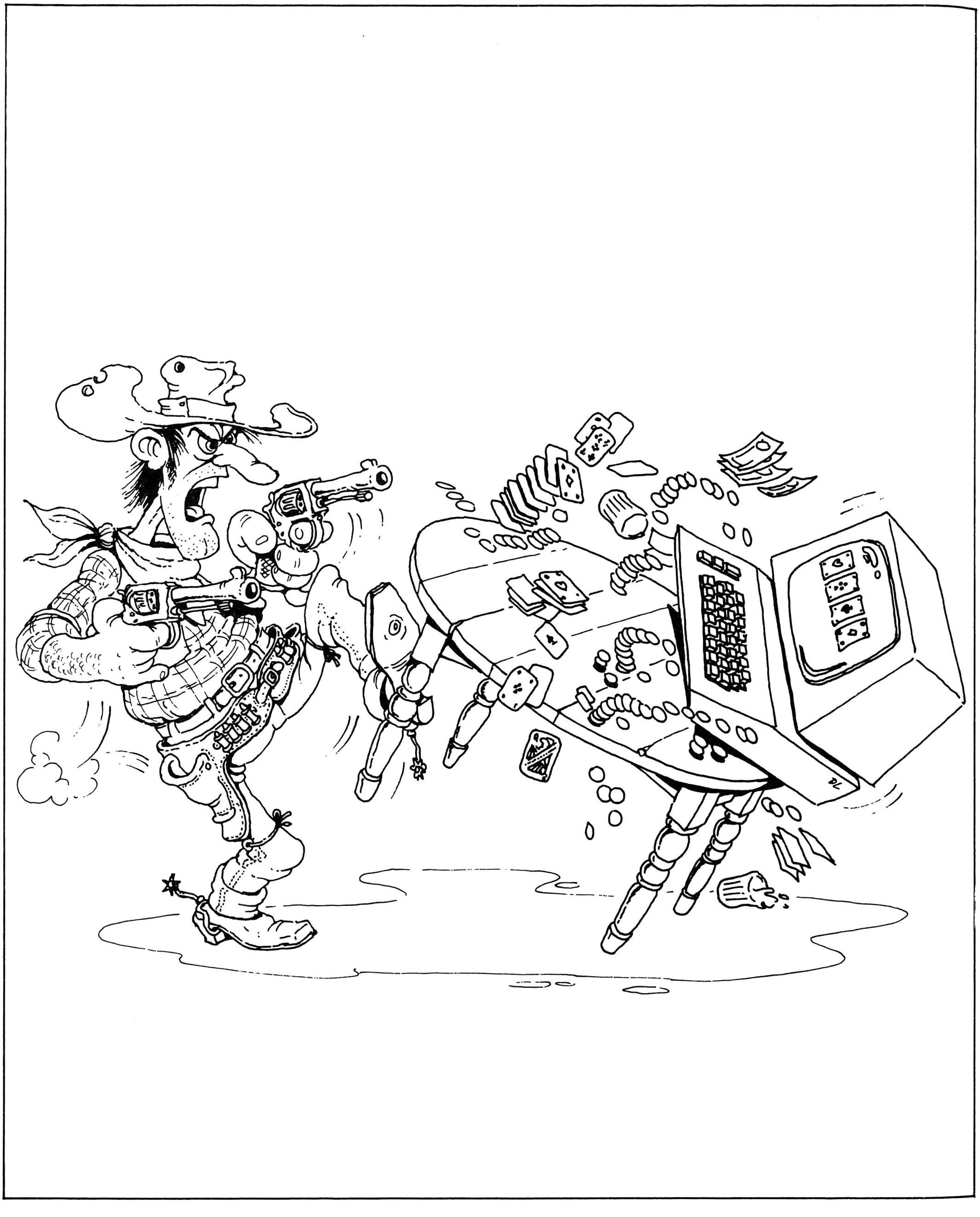
```
(*
 * PICTURE
 * SIMPLY MOVES THE CURSOR WHENEVER YOU
 * PRESS THE BUTTON ON THE PADDLE. THE
 * DIRECTION IS DETERMINED BY THE PADDLE
 * KNOB
 *)
PROGRAM PICTURE (INPUT,OUTPUT,KEYPRESS);
USES TURTLEGRAPHICS, APPLESTUFF;
CONST
   MAXX     = 272;               (* THE LAST X POSITION ON THE SCREEN *)
   MAXY     = 184;               (* THE LAST Y POSITION ON THE SCREEN *)
   NEXTLINE=   7;                (* THE NEXT LINE IS 7 DOTS UP (OR DOWN) *)
   NEXTCH   =   7;               (* THE NEXT CHAR IS 7 BITS AWAY *)
   DOWNANG  = 270;               (* THE ANGLE WHICH IS DOWN TO THE GRAPHICS *)
   UPANG    =  90;               (* THE ANGLE WHICH WILL GO UP *)
   LEFTANG  = 180;               (* TO GO LEFT *)

TYPE
   MOTION   = (UP,DOWN,LEFT,RIGHT); (* WAYS TO MOVE *)

VAR
   DIRECTION : MOTION;           (* DIRECTION CURSOR WILL MOVE *)
   XPOS    : INTEGER;            (* THE CURRENT LOCATION ON THE SCREEN *)
   YPOS    : INTEGER;
   CH      : CHAR;

   (*
    * PRINT
    * PRINT THE CURSOR
    * IF THERE IS SOMETHING AT THE LOCATION,
    * THEN A BLANK WILL BE PRINTED, IF NOT A
    * BLOCK
    *)
   PROCEDURE PRINT;
   VAR
      TODO : BOOLEAN;      (* IF I SHOULD PRINT SOMETHING *)
   BEGIN
     TODO := FALSE;
     CHARTYPE(12);         (* SCREEN IS UNAFFECTED *)
     PENCOLOR(NONE);

     CASE DIRECTION OF

        UP:
              (*
               * CHECK TO SEE IF THE TURTLE IS
               * ON THE SCREEN. IF IT ISN'T THEN
               * DON'T ATTEMPT TO DRAW ANYTHING
               * I.E. DON'T SET TODO TO TRUE
               *)
              IF (TURTLEY < MAXY) AND (TURTLEX < MAXX) THEN
              BEGIN
                 TODO := TRUE;
                 TURNTO(UPANG);
                 MOVE(NEXTLINE)
              END;
      DOWN:
            IF ( TURTLEY > 0 ) AND (TURTLEX < MAXX) THEN
            BEGIN
               TODO := TRUE;
               TURNTO(DOWNANG);
               MOVE(NEXTLINE)
            END;
      LEFT:
            IF TURTLEX > 0 THEN
            BEGIN
               TODO := TRUE;
               TURNTO(LEFTANG);
               MOVE(NEXTCH)
            END;

      RIGHT:
           IF TURTLEX < MAXX THEN
              TODO := TRUE
   END; (* CASE *)

   IF TODO THEN
   BEGIN
      CHARTYPE(3);            (* COMPLEMENT OF ANYTHING THERE *)
      WCHAR(' ');

      IF DIRECTION <> RIGHT THEN
      BEGIN
         TURNTO(LEFTANG);
         MOVE(NEXTCH)
      END
   END
END; (* PRINT *)


(*
 * GETDIR
 * GETS THE DIRECTION TO GO IN NEXT FROM
 * THE PADDLE
 *)
PROCEDURE GETDIR;
CONST
   MAXPAD = 255;              (* THE MAXIMUM NUMBER PADDLE RETURNS *)
   ANG4   =  64;              (* 1/4 THE ABOVE *)

VAR
   PADANG : 0..MAXPAD;        (* THE ANGLE RETURNED BY THE PADDLE *)
BEGIN
   PADANG := PADDLE(0);

   PADANG := PADANG DIV ANG4; (* MAKE IT A NUMBER BETWEEN 0 -> 4 *)

   XPOS := TURTLEX;           (* FIND OUT WHERE WE ARE *)
   YPOS := TURTLEY;
```

```
     CHARTYPE(6);

     CASE PADANG OF

        0 : DIRECTION := DOWN;

        1 : DIRECTION := LEFT;

        2 : DIRECTION := UP;

        3 : DIRECTION := RIGHT
     END;

     MOVETO(0,0);
     CHARTYPE(10);
     CASE DIRECTION OF
        UP : WCHAR('^');

        DOWN : WCHAR('V');

        LEFT : WCHAR('<');

        RIGHT: WCHAR('>')
     END;
     MOVETO(XPOS,YPOS)
  END; (* GETDIR *)
(*
 * MAIN
 * INITS AND WAITS FOR
 * A CHARACTER
 *)
BEGIN
   IF NOT EOLN(INPUT) THEN
      WRITE('PRESS RETURN WHEN READY ');
   READLN(INPUT);
   INITTURTLE;

   WHILE  CH <> ' ' DO
   BEGIN
      GETDIR;           (* DETERMINE DIRECTION *)
      IF BUTTON(0) THEN
         PRINT;
      IF KEYPRESS THEN
      BEGIN
         READ(CH);
         CASE CH OF

            'R': FILLSCREEN(REVERSE);
            'C': FILLSCREEN(BLACK);
            'W': FILLSCREEN(WHITE)
         END
      END
   END;

   TEXTMODE
END.
```

# PART IV

## Cribbage




DL

# CRIBBAGE

**The Game.** In this version of Cribbage, you match your skill and inspiration against the unerring, ever-watchful computer. As is true in many games, the more you learn about your opponent, the more successfully you can compete against it (him? her?). You will find that the characteristics of the computer, directed by this CRIBBAGE program, can work both to your disadvantage and to your advantage.

What the computer is best at, as always, is keeping flawless records of every event that it is directed to watch. This means that no combination of cards (from its own hand or yours) and no sequence of plays will escape its attention and analysis (although you may rest assured that, since it totally lacks guile, it will never take *unfair* advantage of its knowledge of your cards). On the other hand, you, the player (or "the human," as this program likes to call you) will have much greater potential for imaginative strategy than will the computer. What's more, the computer's actions will always be predictable, whereas yours need not be. In short, in this game your opponent the computer is a good, but beatable, Cribbage player; playing this game, you will be challenged to discover the strong points and the weak points in the computer's strategy.

Strategically, the game of Cribbage is divided into three parts; (1) the formation of the crib, (2) the play itself, and (3) the declaration (or "showing") of each hand, with possible subsequent challenges of the declarations. As we will see, a player's strategy is formed during the first and second parts of the game, but points are actually won during the second and third parts.

At the beginning of the game, the "dealer" is chosen by an electronic flip of the coin. (Note, however, that you will never be dealer in more than name only; the computer is a bit domineering on this point—it shuffles and deals the cards for every round, no matter who the "dealer" is.) Six cards are dealt to each opponent. The computer will sort your cards for you in order of their rank (ace, 2, 3, . . . , 10, jack, queen, king), and will display them for you on the screen.

After the deal, each player must select two cards and place them "face down" on the table. These four cards become the crib. You will receive the prompt:

```
REMOVE WHICH CARD?
```

You may identify each of the two cards you wish to place into the crib by typing the rank (A, 2, 3, 4, . . . , T, J, Q, K) and the suit (S, D, C, H for spades, diamonds, clubs, and hearts); however, the rank alone will do if you have no other card in your hand of the same rank.

The computer then puts its two cards down (without telling you what they are, of course), and the crib is complete. The crib belongs to the dealer, which is one reason why the dealer is changed after each round.

*After* the crib is created, the player who is not the dealer cuts the remaining cards of the deck. If it is your turn to cut, you will be asked:

```
CUT WHICH CARD [1-40]?
```

since forty cards remain in the deck. The cut card becomes the COMMON card, or the "starter," and it will be displayed for you on the screen. If the starter is a jack, the dealer immediately scores 1 point (for "his nibbs").

At this point, the actual "play" can begin. Before we describe the play of the cards, however, let us take time out to discuss discard strategy.

The strategy for discarding to the crib can be simple or complex, depending on the hand that has been dealt and on the sophistication of the player who is playing the hand. Each card carries a specific value: the ace is worth 1, the 10, jack, king, and queen are each worth

10, and all the other number cards are worth their face value. At the end of each round you will score from the four cards you choose to keep (out of the six you were dealt) plus the starter. If you are the dealer you will also score from the four-plus-one cards in the crib. (We will see that another kind of scoring occurs during the play itself—before you *declare* your hand—but you begin forming your strategy at the time you discard to the crib.)

You score points from your hand for combinations of fifteen, pairs, runs, or flushes. Each pair is worth 2 points, each combination of fifteen is worth 2 points, and a run (of three or more cards) is worth the number of cards in the run. A flush (four cards of the same suit in the original hand; five cards if the starter is also of the same suit) is worth the number of cards in the flush. At the time the hands are declared, the starter is counted as a fifth card in each of the three hands—the computer's hand, your hand, and the crib.

The tricky part of counting up your hand's score is that you must count *every* combination of pairs, runs, and fifteens. For example, if you have the hand:

```
7H 7D 9D 10C
```

and the starter is the 8 of spades, you have:

- one pair (7H, 7D) worth 2 points
- two runs of four (7H, 8S, 9D, 10C; and 7D, 8S, 9D, 10C) worth 8 points
- two fifteens (7H, 8S; and 7D, 8S) worth 4 points

for a total of 14 points.

In addition, a three-of-a-kind (a "pair royal") is worth 6 points, since it contains three pairs. (For example, 3H, 3D, 3C contains the pairs 3H, 3D; 3H, 3C; and 3D, 3C.) Likewise, a four-of-a-kind (a "double pair royal") is worth 12 points, since it contains six pairs.

Finally, you score 1 point if you have in your hand a jack that is of the same suit as the starter. Keep in mind, though, that you must discard to the crib *before* you know what the starter card will be; thus, the discard strategy is always somewhat tentative and uncertain.

The computer has an unchanging strategy for discarding to the crib. It *keeps* the four cards that leave the maximum possible score in its hand, and it "lays away" the other two cards. This strategy, it turns out, is often the best one, and you may frequently find yourself following it, too. Sometimes, however, a more complicated strategy is called for.

Say you are dealt the following hand in a round when you are not the dealer (and the crib thus belongs to the computer):

```
4C 4D 5S 7D JD QH
```

You have several point combinations: a pair (4C, 4D), and three fifteens (5S, JD; 5S, QH; and 4C, 4D, 7D); but the maximum number of points you can manage to keep is 4: either a pair and a fifteen, or two fifteens. You want to avoid giving points to the "adverse" crib (i.e., a crib that belongs to your opponent). Other dangerous cards to give the crib are 5s, 7s, 8s, and near-runs (cards in sequence or one away from sequence). Thus your hand presents something of a dilemma. Since you want to hold onto your jack and your 5, you might end up discarding a 4 and a 7.

Interestingly enough, given the same hand, the computer would discard the pair of 4s. (This is the first combination it would test, and, since it would never find a *better* combination of points-in-hand, it would stay with its first choice.) For a dealer, the two 4s are perhaps a reasonable discard; for a nondealer, however, it is questionable strategy to give away points to the crib.

The play begins after the crib has been formed and the starter card chosen. (Actually, unknown to you, the computer quickly calculates the point value of each of the three hands—including the starter card in each—before beginning the play. In this calculation, the computer will never err. You can offset this slight advantage by making sure you tally up the points of your own hand—and the crib if you are the dealer—very carefully before proceeding into the play.) If the starter card is a jack, the first message you will see on the screen is

```
I GET A POINT FOR HIS NIBBS
```

or

```
YOU GET A POINT FOR HIS NIBBS
```

depending on who the dealer is. (The computer keeps track of all points and displays the new total whenever points are won by either player. The computer thus takes the place of the cribbage board, which is normally used to count the fast-accumulating points of this game.) The nondealer begins the play.

During the play of cribbage, you and the computer alternately choose a card from your respective hands and place it "face up" on the table. When it is your turn, you will see the prompt:

```
PLAY WHICH? [. . . ]
```

and your remaining cards will be displayed between brackets. When the computer plays a card, you will see the message,

```
I PLAY A . . .
```

with the rank and suit of the computer's card.

Each time a card is played, the card's *value* (from 1 to 10) is added to the current *card count*. This count cannot exceed 31. When one of the players can no longer play a card without exceeding a count of 31, the other player must continue playing cards until he or she (or it) has also reached an unplayable hand. At that point the count returns to zero, and the play begins again with the remaining cards. For example, consider the following sequence:

| | | |
|---|---|---|
| you play 4H | count = 4 | |
| computer plays 4D | count = 8 | 2 points for computer (pair) |
| you play 7S | count = 15 | 2 points for you (fifteen) |
| computer plays 6H | count = 21 | |
| you play 8D | count = 29 | |
| computer cannot play | | |
| you play AS | count = 30 | 1 point for you (last card in series) |
| neither player can play (new series begins) | | |
| computer plays 10H | count = 10 | |

(The play continues until all eight cards—your hand and the computer's hand—have been played.) Notice in this sequence that when the count reached 29, the computer had no card small enough to play; you took a second turn, playing an ace and carrying the count to 30.

There are several ways to win points during the play (some of which are indicated in the round above). For carrying the count to exactly 15 or exactly 31, you receive 2 points. Ending a series short of 31 (i.e., being the last to play before the count returns to zero) is worth 1 point. Playing the last card in a round (of the eight cards in play) is worth 1 point for last, 2 points if it brings the count to 31. Pairs and runs are also worth points, as follows:

- pair—for playing a card of the same rank (*not* value) as the previous card played: 2 points
- three-of-a-kind—for playing the third card (in an uninterrupted sequence) of the same rank: 6 points
- four-of-a-kind—for playing the fourth card (in an uninterrupted sequence) of the same rank: 12 points
- runs—for completing a sequential run of cards: points equal to the number of cards in the run

The round above illustrates points awarded for a pair (2 points), a count of 15 (2 points), and the last card played in a series (1 point).

The computer's strategy during the play is as uncomplicated as its strategy for discarding to the crib. For any given play, it plays the card that will earn the greatest number of points. In a choice between two cards that produce the same number of points, the computer will play the lower ranking card. If it has no card that will give points, the computer simply plays the lowest ranking card in its hand.

The computer makes no attempt to deduce your hand from the cards you play, although obviously you can learn much about the computer's hand by its plays. Thus, your strategy during the play might well be more subtly calculated than the computer's.

When all eight cards have been played, and points have been awarded accordingly, it is time for the players to declare (or "show") their hands. If you are the dealer you will see a display of your own hand plus the starter and the crib's hand plus the starter, and you will be asked:

```
HOW MANY POINTS IN YOUR HAND?
```

and

```
HOW MANY POINTS IN YOUR CRIB?
```

(If you are not the dealer you will only be shown your own hand.) You must enter the *exact* number of points in your hand (or hands) in order to win those points.

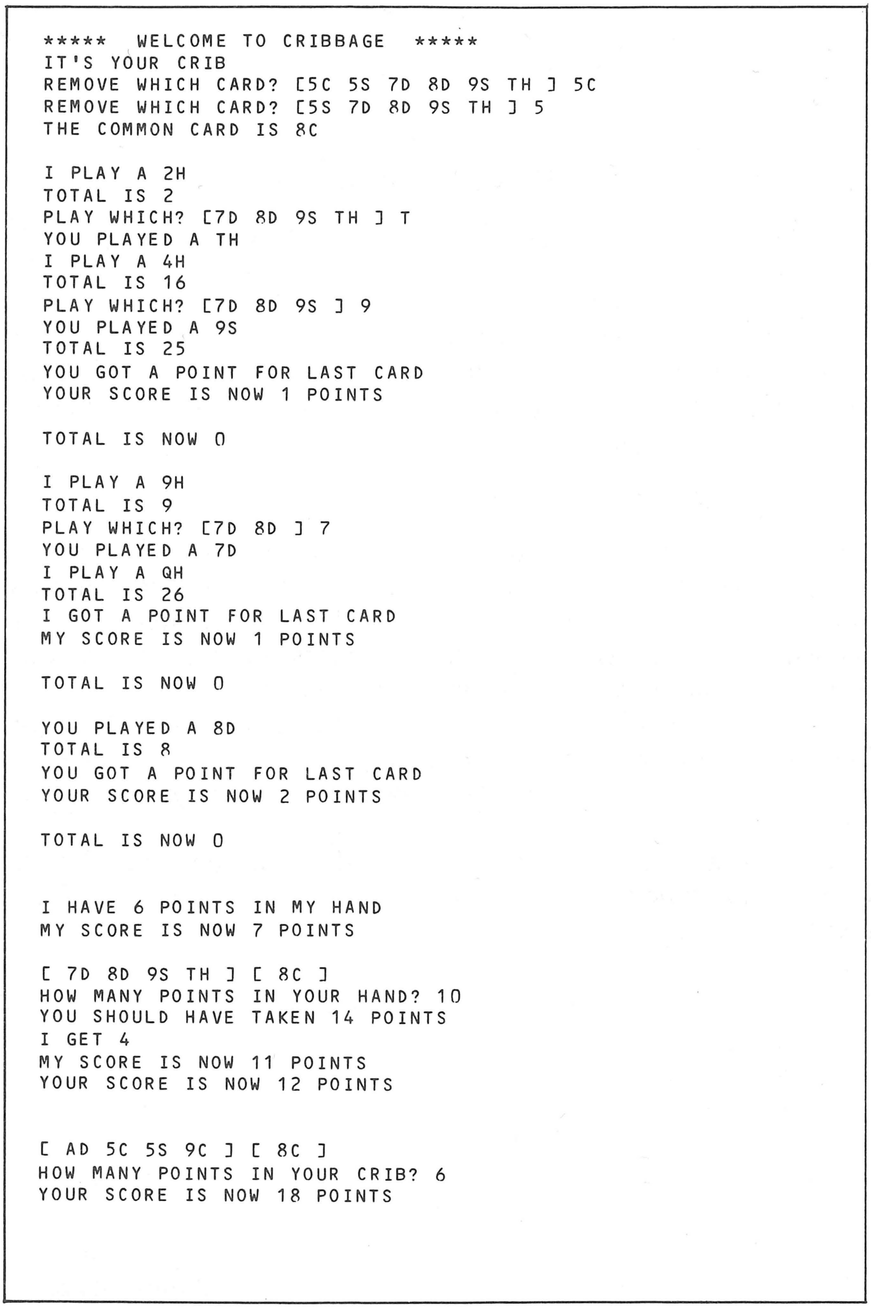
This is the moment the computer has been waiting for: time to show off the task that the computer performs more efficiently than the human. If you overdeclare or underdeclare your points, the computer pounces on you with the message:

```
YOU SHOULD HAVE TAKEN ____ POINTS
```

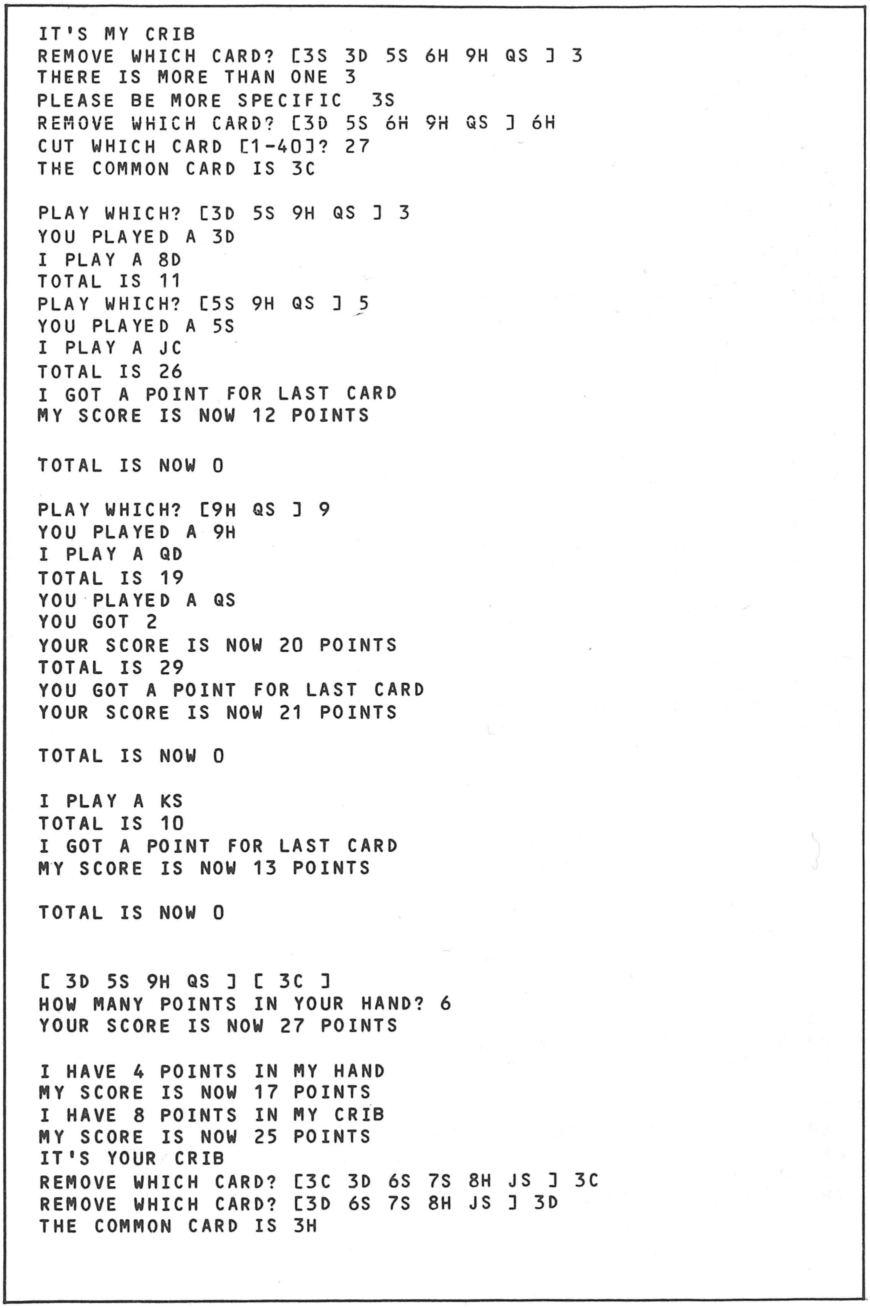
The computer then finds the difference between your declaration and the true number of points contained in your hand and adds this amount to its own score. (In addition, if you *under*declare your points, you only receive the number of points you declare. An overestimate is thus always better than an underestimate.)

The computer will also declare the number of points in its hand (or hands), but you will have no opportunity to challenge its showing (because, of course, the computer is always trustworthy in such matters).

After the showing, the cards are reshuffled and a new set of hands is dealt. The game ends when one of you reaches a total score of 121 points.

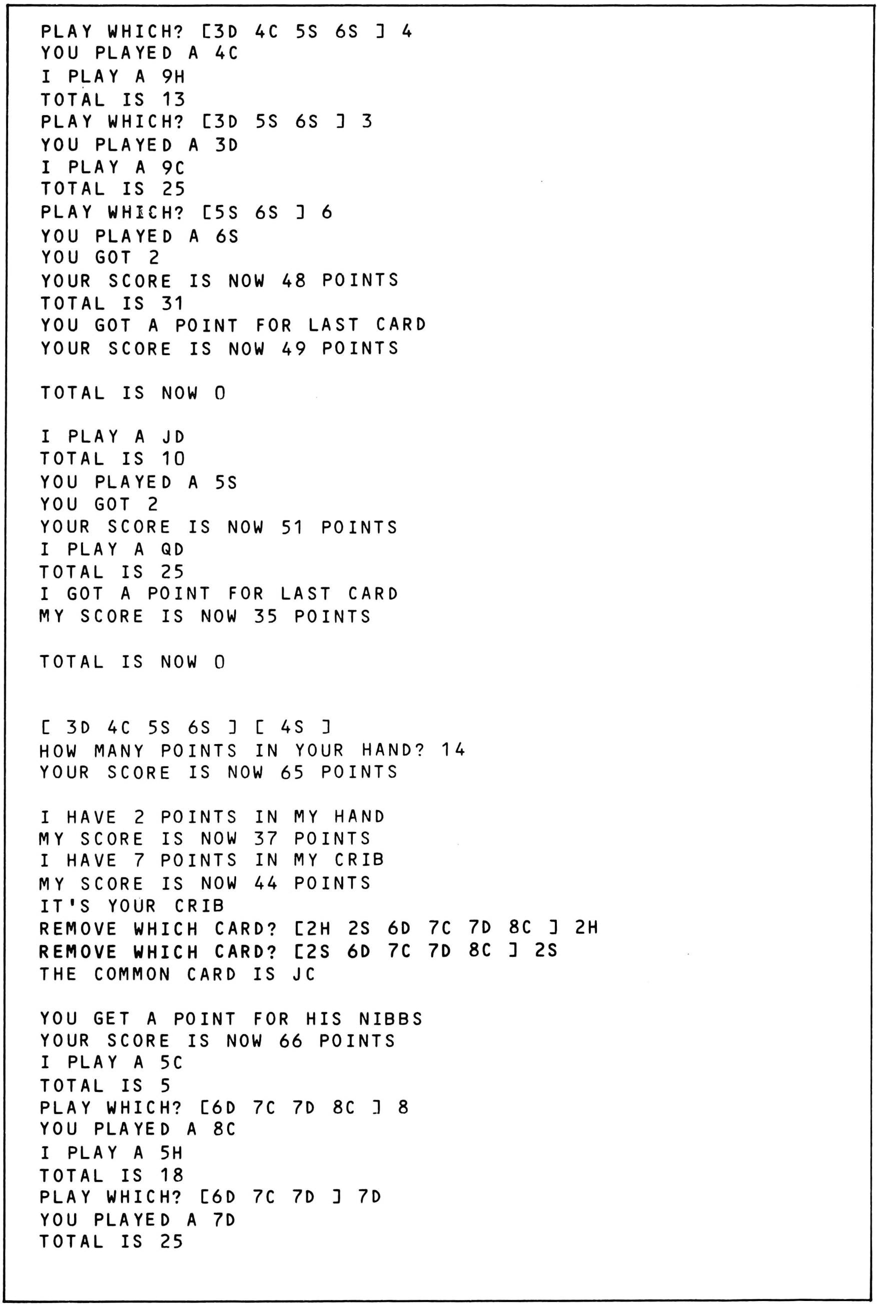


```
*****  WELCOME TO CRIBBAGE  *****
IT'S YOUR CRIB
REMOVE WHICH CARD? [5C 5S 7D 8D 9S TH ] 5C
REMOVE WHICH CARD? [5S 7D 8D 9S TH ] 5
THE COMMON CARD IS 8C

I PLAY A 2H
TOTAL IS 2
PLAY WHICH? [7D 8D 9S TH ] T
YOU PLAYED A TH
I PLAY A 4H
TOTAL IS 16
PLAY WHICH? [7D 8D 9S ] 9
YOU PLAYED A 9S
TOTAL IS 25
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 1 POINTS

TOTAL IS NOW 0

I PLAY A 9H
TOTAL IS 9
PLAY WHICH? [7D 8D ] 7
YOU PLAYED A 7D
I PLAY A QH
TOTAL IS 26
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 1 POINTS

TOTAL IS NOW 0

YOU PLAYED A 8D
TOTAL IS 8
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 2 POINTS

TOTAL IS NOW 0


I HAVE 6 POINTS IN MY HAND
MY SCORE IS NOW 7 POINTS

[ 7D 8D 9S TH ] [ 8C ]
HOW MANY POINTS IN YOUR HAND? 10
YOU SHOULD HAVE TAKEN 14 POINTS
I GET 4
MY SCORE IS NOW 11 POINTS
YOUR SCORE IS NOW 12 POINTS


[ AD 5C 5S 9C ] [ 8C ]
HOW MANY POINTS IN YOUR CRIB? 6
YOUR SCORE IS NOW 18 POINTS
```



```
IT'S MY CRIB
REMOVE WHICH CARD? [3S 3D 5S 6H 9H QS ] 3
THERE IS MORE THAN ONE 3
PLEASE BE MORE SPECIFIC  3S
REMOVE WHICH CARD? [3D 5S 6H 9H QS ] 6H
CUT WHICH CARD [1-40]? 27
THE COMMON CARD IS 3C

PLAY WHICH? [3D 5S 9H QS ] 3
YOU PLAYED A 3D
I PLAY A 8D
TOTAL IS 11
PLAY WHICH? [5S 9H QS ] 5
YOU PLAYED A 5S
I PLAY A JC
TOTAL IS 26
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 12 POINTS

TOTAL IS NOW 0

PLAY WHICH? [9H QS ] 9
YOU PLAYED A 9H
I PLAY A QD
TOTAL IS 19
YOU PLAYED A QS
YOU GOT 2
YOUR SCORE IS NOW 20 POINTS
TOTAL IS 29
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 21 POINTS

TOTAL IS NOW 0

I PLAY A KS
TOTAL IS 10
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 13 POINTS

TOTAL IS NOW 0


[ 3D 5S 9H QS ] [ 3C ]
HOW MANY POINTS IN YOUR HAND? 6
YOUR SCORE IS NOW 27 POINTS

I HAVE 4 POINTS IN MY HAND
MY SCORE IS NOW 17 POINTS
I HAVE 8 POINTS IN MY CRIB
MY SCORE IS NOW 25 POINTS
IT'S YOUR CRIB
REMOVE WHICH CARD? [3C 3D 6S 7S 8H JS ] 3C
REMOVE WHICH CARD? [3D 6S 7S 8H JS ] 3D
THE COMMON CARD IS 3H
```

```
I PLAY A 5H
TOTAL IS 5
PLAY WHICH? [6S 7S 8H JS ] J
YOU PLAYED A JS
YOU GOT 2
YOUR SCORE IS NOW 29 POINTS
I PLAY A TD
TOTAL IS 25
YOU PLAYED A 6S
YOU GOT 2
YOUR SCORE IS NOW 31 POINTS
TOTAL IS 31
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 32 POINTS

TOTAL IS NOW 0

I PLAY A QD
TOTAL IS 10
PLAY WHICH? [7S 8H ] 8
YOU PLAYED A 8H
I PLAY A QH
TOTAL IS 28
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 26 POINTS

TOTAL IS NOW 0

YOU PLAYED A 7S
TOTAL IS 7
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 33 POINTS

TOTAL IS NOW 0


I HAVE 8 POINTS IN MY HAND
MY SCORE IS NOW 34 POINTS

[ 6S 7S 8H JS ] [ 3H ]
HOW MANY POINTS IN YOUR HAND? 5
YOUR SCORE IS NOW 38 POINTS


[ 3C 3D 4S 6D ] [ 3H ]
HOW MANY POINTS IN YOUR CRIB? 8
YOUR SCORE IS NOW 46 POINTS

IT'S MY CRIB
REMOVE WHICH CARD? [2D 3D 4C 5S 6S TC ] 2
REMOVE WHICH CARD? [3D 4C 5S 6S TC ] T
CUT WHICH CARD [1-40]? 20
THE COMMON CARD IS 4S
```



```
PLAY WHICH? [3D 4C 5S 6S ] 4
YOU PLAYED A 4C
I PLAY A 9H
TOTAL IS 13
PLAY WHICH? [3D 5S 6S ] 3
YOU PLAYED A 3D
I PLAY A 9C
TOTAL IS 25
PLAY WHICH? [5S 6S ] 6
YOU PLAYED A 6S
YOU GOT 2
YOUR SCORE IS NOW 48 POINTS
TOTAL IS 31
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 49 POINTS

TOTAL IS NOW 0

I PLAY A JD
TOTAL IS 10
YOU PLAYED A 5S
YOU GOT 2
YOUR SCORE IS NOW 51 POINTS
I PLAY A QD
TOTAL IS 25
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 35 POINTS

TOTAL IS NOW 0


[ 3D 4C 5S 6S ] [ 4S ]
HOW MANY POINTS IN YOUR HAND? 14
YOUR SCORE IS NOW 65 POINTS

I HAVE 2 POINTS IN MY HAND
MY SCORE IS NOW 37 POINTS
I HAVE 7 POINTS IN MY CRIB
MY SCORE IS NOW 44 POINTS
IT'S YOUR CRIB
REMOVE WHICH CARD? [2H 2S 6D 7C 7D 8C ] 2H
REMOVE WHICH CARD? [2S 6D 7C 7D 8C ] 2S
THE COMMON CARD IS JC

YOU GET A POINT FOR HIS NIBBS
YOUR SCORE IS NOW 66 POINTS
I PLAY A 5C
TOTAL IS 5
PLAY WHICH? [6D 7C 7D 8C ] 8
YOU PLAYED A 8C
I PLAY A 5H
TOTAL IS 18
PLAY WHICH? [6D 7C 7D ] 7D
YOU PLAYED A 7D
TOTAL IS 25
```

```
YOU PLAYED A 6D
YOU GOT 2
YOUR SCORE IS NOW 68 POINTS
TOTAL IS 31
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 69 POINTS

TOTAL IS NOW 0

I PLAY A TS
TOTAL IS 10
YOU PLAYED A 7C
I PLAY A TH
TOTAL IS 27
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 45 POINTS

TOTAL IS NOW 0


I HAVE 16 POINTS IN MY HAND
MY SCORE IS NOW 61 POINTS

[ 6D 7C 7D 8C ] [ JC ]
HOW MANY POINTS IN YOUR HAND? 12
YOUR SCORE IS NOW 81 POINTS


[ 2H 2S 3S QD ] [ JC ]
HOW MANY POINTS IN YOUR CRIB? 10
YOUR SCORE IS NOW 91 POINTS

IT'S MY CRIB
REMOVE WHICH CARD? [AH 5H 6H 8S QC KS ] 1
ILLEGAL INPUT
WHICH CARD? A
REMOVE WHICH CARD? [5H 6H 8S QC KS ] 6
CUT WHICH CARD [1-40]? 23
THE COMMON CARD IS 9S

PLAY WHICH? [5H 8S QC KS ] 8
YOU PLAYED A 8S
I PLAY A 6D
TOTAL IS 14
PLAY WHICH? [5H QC KS ] K
YOU PLAYED A KS
TOTAL IS 24
YOU PLAYED A 5H
TOTAL IS 29
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 92 POINTS

TOTAL IS NOW 0

I PLAY A JH
TOTAL IS 10
```

```
YOU PLAYED A QC
I PLAY A QD
I GOT 2
MY SCORE IS NOW 63 POINTS
TOTAL IS 30
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 64 POINTS

TOTAL IS NOW 0

I PLAY A KC
TOTAL IS 10
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 65 POINTS

TOTAL IS NOW 0


[ 5H 8S QC KS ] [ 9S ]
HOW MANY POINTS IN YOUR HAND? 4
YOUR SCORE IS NOW 96 POINTS

I HAVE 5 POINTS IN MY HAND
MY SCORE IS NOW 70 POINTS
I HAVE 4 POINTS IN MY CRIB
MY SCORE IS NOW 74 POINTS
IT'S YOUR CRIB
REMOVE WHICH CARD? [2D 5S 6C 9D 9H QD ] 5
REMOVE WHICH CARD? [2D 6C 9D 9H QD ] Q
THE COMMON CARD IS 5D

I PLAY A 7D
TOTAL IS 7
PLAY WHICH? [2D 6C 9D 9H ] 9H
YOU PLAYED A 9H
I PLAY A 9C
I GOT 2
MY SCORE IS NOW 76 POINTS
TOTAL IS 25
PLAY WHICH? [2D 6C 9D ] 9D
SORRY, THAT IS MORE THAN 31
PLAY WHICH? [2D 6C 9D ] 6
YOU PLAYED A 6C
YOU GOT 2
YOUR SCORE IS NOW 98 POINTS
TOTAL IS 31
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 99 POINTS

TOTAL IS NOW 0

I PLAY A 8D
TOTAL IS 8
PLAY WHICH? [2D 9D ] 9
YOU PLAYED A 9D
```

```
I PLAY A TC
TOTAL IS 27
YOU PLAYED A 2D
TOTAL IS 29
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 100 POINTS

TOTAL IS NOW 0


I HAVE 8 POINTS IN MY HAND
MY SCORE IS NOW 84 POINTS

[ 2D 6C 9D 9H ] [ 5D ]
HOW MANY POINTS IN YOUR HAND? 6
YOUR SCORE IS NOW 106 POINTS


[ 2C 2S 5S QD ] [ 5D ]
HOW MANY POINTS IN YOUR CRIB? 8
YOUR SCORE IS NOW 114 POINTS

IT'S MY CRIB
REMOVE WHICH CARD? [AH 6D 7C 7H QH KH ] A
REMOVE WHICH CARD? [6D 7C 7H QH KH ] 6
CUT WHICH CARD [1-40]? 27
THE COMMON CARD IS 9D

PLAY WHICH? [7C 7H QH KH ] 7C
YOU PLAYED A 7C
I PLAY A JS
TOTAL IS 17
PLAY WHICH? [7H QH KH ] Q
YOU PLAYED A QH
TOTAL IS 27
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 115 POINTS

TOTAL IS NOW 0

I PLAY A QS
TOTAL IS 10
PLAY WHICH? [7H KH ] 7
YOU PLAYED A 7H
I PLAY A QD
TOTAL IS 27
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 85 POINTS

TOTAL IS NOW 0

YOU PLAYED A KH
I PLAY A KS
I GOT 2
```

```
MY SCORE IS NOW 87 POINTS
TOTAL IS 20
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 88 POINTS

TOTAL IS NOW 0


[ 7C 7H QH KH ] [ 9D ]
HOW MANY POINTS IN YOUR HAND? 2
YOUR SCORE IS NOW 117 POINTS

I HAVE 8 POINTS IN MY HAND
MY SCORE IS NOW 96 POINTS
I HAVE 4 POINTS IN MY CRIB
MY SCORE IS NOW 100 POINTS
IT'S YOUR CRIB
REMOVE WHICH CARD? [2D 4D 6S 8H JS QS ] 4
REMOVE WHICH CARD? [2D 6S 8H JS QS ] 6
THE COMMON CARD IS 4H

I PLAY A 5D
TOTAL IS 5
PLAY WHICH? [2D 8H JS QS ] J
YOU PLAYED A JS
YOU GOT 2
YOUR SCORE IS NOW 119 POINTS
I PLAY A JC
I GOT 2
MY SCORE IS NOW 102 POINTS
TOTAL IS 25
YOU PLAYED A 2D
TOTAL IS 27
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 120 POINTS

TOTAL IS NOW 0

I PLAY A TS
TOTAL IS 10
PLAY WHICH? [8H QS ] Q
YOU PLAYED A QS
I PLAY A QH
I GOT 2
MY SCORE IS NOW 104 POINTS
TOTAL IS 30
I GOT A POINT FOR LAST CARD
MY SCORE IS NOW 105 POINTS

TOTAL IS NOW 0

YOU PLAYED A 8H
TOTAL IS 8
YOU GOT A POINT FOR LAST CARD
YOUR SCORE IS NOW 121 POINTS
CONGRATULATIONS, YOU HAVE BEATEN THE COMPUTER
```

# The Program.

Even though this Cribbage program is long, you will probably find it to be very readable. The many comments, combined with meaningful procedure and variable names, allow you to find your way through this program with relative ease.

The following paragraphs will describe the action of this program from beginning to end. Each procedure or function will be studied at the point where it is first called in the program. (Referring frequently to the structure chart for this program will help you keep track of the relationship between one procedure and the next.)

**The main program** block displays a welcome message and calls RANDOMIZE. It then calls procedure MAKEDECK; it initializes the variables HSCORE (the human's score) and CSCORE (the computer's score) to zero; and it calls procedure PLAY.

**Procedure MAKEDECK** initializes the variable DECK. DECK is an array of CARD; CARD is a record type with two fields, RANK and SUIT, as defined in the program heading:

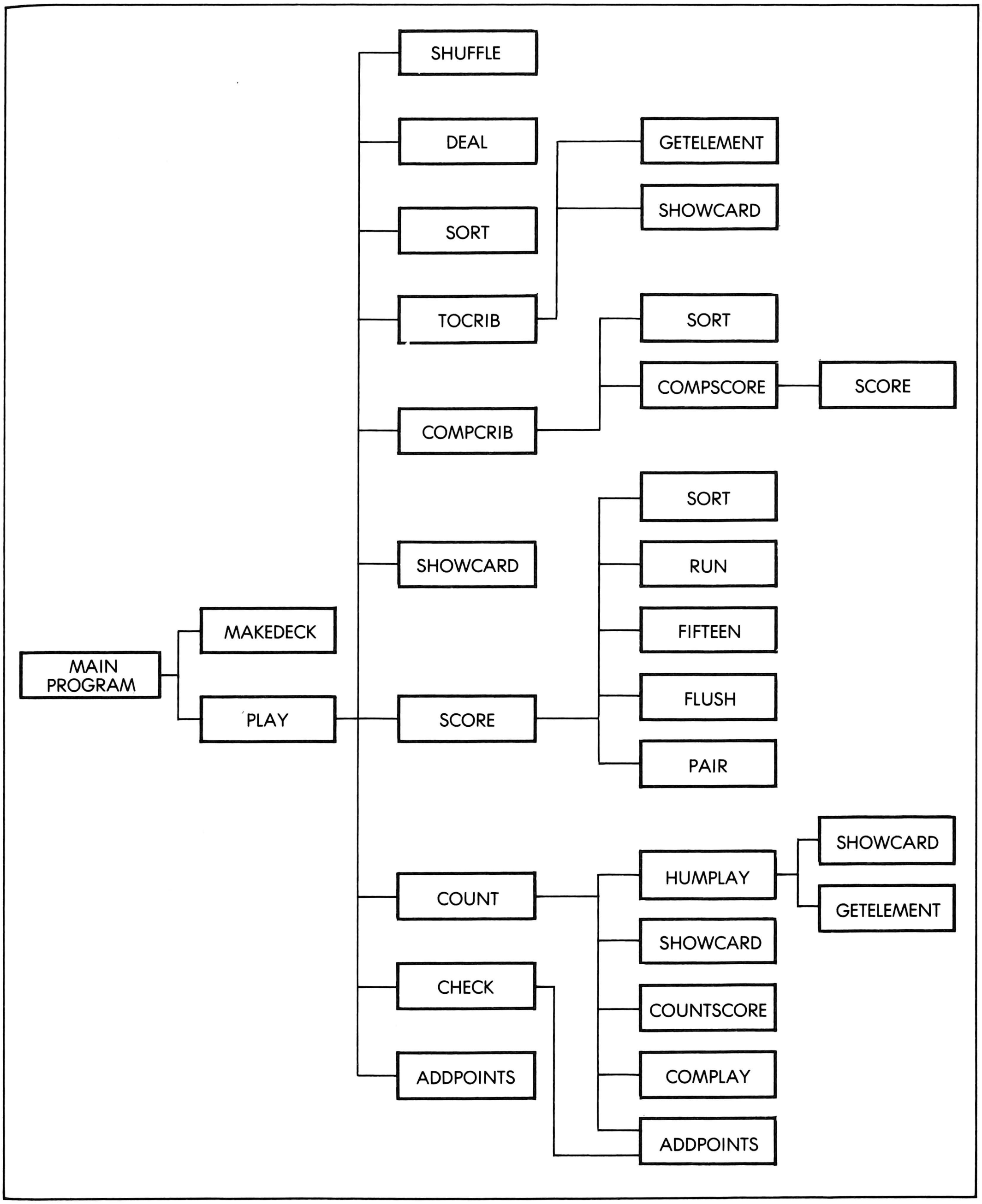
```
CARD = RECORD
  RANK: RANKTYPE;
  SUIT: SUITTYPE
END;
```

DECK is initialized inside two levels of **FOR** loops, using CARDNUM as the array index. The outer loop controls the SUIT and the inner loop controls the RANK, so that DECK[1] to DECK[13] contain all the HEARTS, DECK[14] to DECK[26] contain the DIAMONDS, and so on.

**The Main Program (continued).** PLAY is called with a BOOLEAN expression as its parameter:

```
PLAY((RANDOM MOD 2) = 0)
```

This expression represents the flip of the coin to see who will be the first dealer. Since RANDOM **MOD** 2 produces either a 0 or a 1, the expression has an equal chance of being TRUE or FALSE. The actual BOOLEAN value is passed to the formal parameter WHO in PLAY.


MAIN PROGRAM
MAKEDECK
PLAY
SHUFFLE
DEAL
SORT
TOCRIB
COMPCRIB
SHOWCARD
SCORE
COUNT
CHECK
ADDPOINTS
GETELEMENT
SHOWCARD
SORT
COMPSCORE
SCORE
SORT
RUN
FIFTEEN
FLUSH
PAIR
HUMPLAY
SHOWCARD
GETELEMENT
SHOWCARD
COUNTSCORE
COMPLAY
ADDPOINTS

As we will see, several events throughout the program depend on the value of WHO. These events are summarized in the following table:

| *TRUE* | *FALSE* |
|---|---|
| crib belongs to computer | crib belongs to player |
| player cuts deck | computer cuts deck |
| computer receives point for HIS NIBBS | player receives point for HIS NIBBS |
| player starts play | computer starts play |
| computer "shows" crib | player "shows" crib |

The value of WHO is switched at the end of PLAY (thus alternating the dealer):

```
WHO := NOT WHO
```

**Procedure PLAY** controls the action of the CRIBBAGE game. The main block of PLAY contains one long **WHILE** loop that begins:

```
WHILE TRUE DO
```

This looks suspiciously like an infinite loop, and in fact it would be, except for a statement that we will look at later on in procedure ADDPOINTS. (PLAY is ended via the UCSD function EXIT when one of the players' scores reaches 121.) For now we can see that this **WHILE** loop continues to deal new hands until the appropriate time comes to end the game.

Play starts out by calling SHUFFLE and DEAL.

**Procedure SHUFFLE** uses the RANDOM function and the variable TMP (of type CARD) to shuffle the 52 cards. The algorithm it follows looks a lot like a sort routine, but instead of putting the cards in order, it is taking them out of order.

**Procedure DEAL** gives six cards each to the player and the computer by initializing the arrays HUMAN and COMP. Both of these variables are of type HANDTYPE, which is declared in the program heading:

```
HANDTYPE = ARRAY[1..DEALSIZE] OF CARD;
```

DEALSIZE is a constant (equal to 6) and CARD is the record type that we have already seen in the initialization of DECK.

In this deal, HUMAN is given the first six cards of DECK, and COMP is given cards seven to twelve. (This is not quite standard procedure in Cribbage, but since we have confidence in the shuffling process, we can expect this to be a fair deal.)

DEAL also initializes the rank of the starter card to zero:

```
COMMON.RANK := 0
```

We will see the importance of this when we first look at procedure SCORE.

**PLAY (continued).** Next a message is displayed to identify the dealer. One of two messages is written, depending on the value of WHO. Notice the use of the two single quotes in the WRITELN statements:

```
WRITELN('IT''S YOUR CRIB')
```

This produces the message:

```
IT'S YOUR CRIB
```

Following these WRITELN statements are four procedure calls. The first is to SORT, to put the human player's cards in order. Then come TOCRIB (for the human to discard to the crib) and COMPCRIB (for the computer to discard to the crib). Finally, another call to SORT puts the crib's cards in order. Let us examine each of these procedures.

**Procedure SORT** receives the number of cards to be sorted (N) and the actual cards to be sorted (HAND). It sorts the cards in ascending order, using a "bubble sort" algorithm. It does the job in two levels of loops. The inner **FOR** loop passes through the cards and exchanges sequential pairs of cards that are out of order. The outer **REPEAT/UNTIL** loop continues the passes until an entire pass is made without changing a card. (The BOOLEAN variable TOUCHED is used to test the action of the inner loop.)

**Procedure TOCRIB** begins by displaying the prompt:

```
REMOVE WHICH CARD?
```

and then displays the human player's entire hand, card by card, through repeated calls to procedure SHOWCARD. Each time SHOWCARD is called it is sent a different element of the array HUMAN.

**Procedure SHOWCARD** receives the card information in the formal parameter TOSHOW (a record variable of type CARD). SHOWCARD contains two **CASE** statements to display first the rank and then the suit of the card it is passed. This procedure is called frequently throughout the program.

**TOCRIB (continued).** Function GETELEMENT is called next to prompt the player to choose a card to place into the crib. (This function is also called later in the program, whenever the human player must be prompted to enter a card.)

**Function GETELEMENT** begins by reading the player's input. Through use of the standard function EOLN, the read statements are designed so that either one or two characters can be read (the rank only, or the rank and the suit together):

```
READ(IRANK);
IF NOT EOLN THEN
   READLN(ISUIT)
ELSE
   READLN;
```

Then the input characters are validated using the set operator **IN**. If something is wrong with the input, an error message is displayed and GETELEMENT is called recursively:

```
GETELEMENT := GETELEMENT
```

Once valid data have been input, two **CASE** statements are used to assign appropriate values to the variables RANK (of type RANKTYPE) and SUIT (of type SUITYPE). Notice the use of ORD to assign numeric values to RANK for cards '2' through '9':

```
'2', '3', '4', '5', '6', '7', '8', '9' :
                RANK := ORD(IRANK) - ORD('0');
```

The remainder of function GETELEMENT compares each value of HUMAN with RANK and SUIT to make sure that the player has identified one and only one card that is currently in HUMAN. One of the first things we are reminded of in looking at this routine is that Pascal allows a variable to have the same name as a record field; for example:

```
IF HUMAN[INDEX].RANK = RANK THEN
```

and

```
IF HUMAN[INDEX].SUIT = SUIT THEN
```

The variable MANY is the key to understanding the action of the **FOR** loop at the beginning of this routine. If, on leaving the **FOR** loop, MANY equals 0, then no card has been found corresponding to the player's input. If MANY is greater than 1, on the other hand, the player's input did not adequately identify a single card in the hand. (For example, if the player has both a 4 of diamonds and a 4 of hearts, then '4' alone is not enough information to identify a card.)

If MANY equals 1 or −5, however, the player's input is valid; 1 means that the player correctly input only the rank of a card, and −5 means that the player correctly input both the rank and the suit of a card. In either case GETELEMENT returns the index of the card (in HUMAN) that the player wishes to place in the crib.

**TOCRIB (continued).** The tasks that remain, once a card has been chosen, are first to assign the card to the crib:

```
CRIB[NUMGONE + 1] := HUMAN[CARDNUM]
```

and then to remove the card from HUMAN and close up the array, which is done in a **WHILE** loop at the end of TOCRIB.

Notice that the action of TOCRIB is controlled by a **FOR** loop; the loop index NUMGONE increments from 0 to 1. Thus the player places *two* cards in the crib.

**Procedure COMPCRIB** is called next from PLAY to give the computer its turn to discard to the crib. COMPCRIB and its function COMPSCORE contain the computer's strategy for discarding. The variable that is essential to understanding this procedure is BEST, which is a record of type BESTREC:

```
BESTREC = RECORD
  POINTS: INTEGER;
  FIRST, SECOND: 1..DEALSIZE
END;
```

COMPCRIB begins by initializing BEST.POINTS to −1 and then calling SORT to arrange the computer's hand. Then, within two levels of **FOR** loops, every possible combination of two cards is removed, in turn, from the hand; the point yield of the remaining four cards is calculated by a call to COMPSCORE. Each time COMPSCORE returns a greater number of points than the previous points recorded (IF POINTS > BEST.POINTS . . . ) the two discard cards (I and J) and the resulting number of points are stored in BEST:

```
BEST.POINTS := POINTS;
BEST.FIRST := I;
BEST.SECOND := J
```

Thus, at the end of the looping, BEST contains the computer's discard information.

**Function COMPSCORE** uses three **FOR** loops (one after the other) to put the four cards that are to be tested into the first four elements of the array TMP. COMPSCORE then calls SCORE (passing TMP as a parameter) to calculate the number of points in the hand.

**Function SCORE** receives TMP in the formal parameter HAND. SCORE uses the variable RETURN to accumulate the total point value of the hand. RETURN is accordingly initialized to zero at the beginning of the main block of the function.

SCORE is actually used in two different contexts. The first situation is the present one: the computer's discard strategy is being determined. The second situation, which we will see shortly, is the calculation of the actual value of the three hands in the game—COMP (the computer's hand), HUMAN (the player's hand), and CRIB (the crib's hand). The first time SCORE is called, only four cards are "passed" to it (in TMP) because the computer is searching for its four most valuable cards. In subsequent calls to SCORE, however, five cards will be involved, because COMP, HUMAN, and CRIB will each contain the "starter" card, COMMON. (The local variable CARDMAX, set to either 4 or 5, indicates how many cards are in the hand that SCORE is investigating.)

Thus, SCORE must check COMMON.RANK to see if the starter card has been chosen yet. Recall that COMMON.RANK was initialized to 0 in procedure DEAL. If there is a starter card, SCORE sets

CARDMAX to 5, and examines the hand in question to see if it contains a jack of the same suit as the starter. If so, the hand scores 1 point (RETURN := 1). The starter card is placed into the hand and the new hand (of five cards) is sorted:

```
HAND[5] := COMMON;
SORT(5,HAND)
```

Otherwise, if the starter card has not been chosen yet (i.e., if COMMON.RANK still has the value 0, which is the case in the present call to SCORE), the above action is skipped. Four functions are called to count up the different scoring combinations—FIFTEEN, PAIR, RUN, and FLUSH. Each returned value is added to RETURN. (RETURN is, of course, the value that SCORE returns to its calling procedure.)

**FIFTEEN, RUN, and PAIR** each investigate the CARDMAX (i.e., 4 or 5) cards in the HAND passed by SCORE.

FIFTEEN uses four levels of nested loops to count up the different combinations of fifteen in the hand, accumulating the number of combinations in the variable WAYS. WAYS is finally multiplied by 2 to produce the point value of the fifteens. Note the use of the **WITH** statement in FIFTEEN:

```
WITH HAND[I] DO
```

This statement allows a shorter notation for references to the fields of HAND; the field names can be used alone, detached from the record name. (The **WITH** statement may also improve computation efficiency.)

Function RUN goes through all the cards, checking for runs and multiple runs. The variable MATCHED accumulates the number (minus 1) of cards in a run, and MULT accumulates the multiples of a single run. (for example, the combination 3S, 4D, 4H, 5S actually contains the *two* runs 3S, 4D, 5S, and 3S, 4H, 5S; thus MULT would equal 2 for this combination.)

Function PAIR uses the variable MATCHED to find pairs, pair royals, and double pair royals, and scores accordingly via two **CASE** statements.

**Function FLUSH** checks through the original hand to see how many cards are of the same suit. (The number is accumulated in RETURN.) If it finds a flush, it then checks the starter card to see if it is also of the same suit. (Notice that FLUSH must be called *before* the starter card—COMMON—is sorted into the original hand.) FLUSH returns a 4 or a 5 if a flush exists, or a 0 if there is no flush.

**COMPCRIB (continued).** Once the best four-card scoring combination has been found for the computer's hand, COMPCRIB places those four cards into the first four elements of COMP, and uses the FIRST and SECOND fields of BEST to reference the cards that are to be placed in the crib:

```
CRIB[3] := COMP[BEST.FIRST];
CRIB[4] := COMP[BEST.SECOND];
```

**PLAY (continued).** The next step is to sort the crib, now that it contains all four of its cards:

```
SORT (4, CRIB);
```

Finally it is time to cut the remaining deck to determine the starter (COMMON) card. If the computer is the dealer (WHO = TRUE), the player cuts. A prompt is displayed and the player enters a number from 1 to 40. The starter card is determined by adding 12 (the number of cards that are out of the deck) to the player's input, USER:

```
COMMON := DECK[USER + 12]
```

If the human player is the dealer, on the other hand, the computer somewhat cavalierly cuts the deck and assigns a value to COMMON in a single statement, using RANDOM and **MOD**:

```
COMMON := DECK[13 + RANDOM MOD 40]
```

A call to procedure SHOWCARD then displays the COMMON card, and three quick calls to the SCORE function tally up the point value of each hand, assigning the values to CPOINTS, HPOINTS, and CRBPNTS.

Now the program is ready to initiate the actual *play* sequence of the game; to do so it calls procedure COUNT.

**Procedure COUNT** starts out with a series of initializations, after which it checks to see if the starter is a jack:

```
IF COMMON.RANK = 11 THEN . . .
```

If it is, then the dealer (identified by the value WHO) is awarded a point. A message to this effect is displayed, and procedure ADDPOINTS is called. Note that this is the first time a point has actually been "pegged." The point value of the hands has been counted up, but the points have not been awarded yet.

**Procedure ADDPOINTS** will be called many times from here to the end of the program. ADDPOINTS's formal parameters are WHO, indicating which player is to score, and AMOUNT, indicating the number of points to score. The procedure heading for ADDPOINTS can be found just above function GETELEMENT. The heading contains a FORWARD declaration, because the procedure block itself is located after procedure PLAY.

ADDPOINTS simply increases HSCORE or CSCORE by the value of AMOUNT and displays the new total score. It also ends the game if one of the scores is greater than or equal to WINPOINTS. For example:

```
IF HSCORE >= WINPOINTS THEN
   EXIT(PLAY)
```

The use of the EXIT procedure in this case greatly simplifies the termination of procedure PLAY and the program.

**COUNT (continued).** COUNT then makes a copy of the human player's entire hand in OLDHUMAN. (This is done so that the player's hand can be displayed again for the "showing" at the end of the round.) Nearly the rest of COUNT is controlled by a **WHILE** loop that continues the play until both players are out of cards:

```
WHILE (HUMLEFT > 0) OR (COMLEFT > 0) DO
```

Once again, depending on the value of WHO, either the computer or the human plays first. Notice that the first line in each of the next two blocks changes the value of WHO:

```
WHO := FALSE;
```

indicating the computer goes next, or:

```
WHO := TRUE;
```

indicating the human goes next.

For the human's turn, a call is made to function HUMPLAY.

**Function HUMPLAY** first tests to see if the human can play at all.

It does this by assigning the rank of the first card in HUMAN to I. Then, if

```
I + CURCOUNT > 31
```

the human cannot play, and the function is terminated. Recall that the cards are sorted in ascending order; thus if the first, and lowest-ranking card would carry the count, CURCOUNT, beyond 31, then the player has no playable card.

If the first card is playable, HUMCOUNT performs a similar test on the second card. If the second card turns out not to be playable, HUMPLAY selects the first card and terminates. The rationale here is that there is no sense giving the player a choice if he or she has only one playable card.

Finally, if there is a choice of playable cards in the human's hand, HUMPLAY displays all the cards (through calls to SHOWCARD) and prompts the player to select a card (a call to GETELEMENT). Never trustful of a human, the computer tests the chosen card to make sure it is actually playable:

```
IF (J + CURCOUNT) > 31 THEN . . .
```

and displays a polite error message if the player must choose again:

```
SORRY, THAT IS MORE THAN 31
```

**COUNT (continued).** If the human was able to play, COUNT next displays the chosen card (a call to SHOWCARD) and calls COUNTSCORE to see how many points the play is worth.

**Function COUNTSCORE** places the played card (passed to the formal parameter NEWCARD) into CNTHAND. CNTHAND is defined in the procedure heading of COUNT:

```
CNTHAND: ARRAY[1..8] OF CARD;
```

It then increases CURCOUNT by the value (1 through 10) of the new card. If the new card was the first card to be played in a series (CNTNUM = 1) then the function is terminated; there is no score to count. Otherwise COUNTSCORE begins looking for scoring combinations and accumulating the points in RETURN. It first tests for a CURCOUNT of 15 or 31 (worth 2 points); then it searches for pairs and runs, and scores accordingly.

An interesting exercise is to compare these search algorithms to those used in SCORE. The situations are different: SCORE looks through an entire hand to find points, whereas COUNTSCORE need only look backwards through CNTHAND to find the points of a single play.

**COUNT (continued).** If the player made any points, they are displayed and added to HSCORE by a call to ADDPOINTS. Then the card that was played is removed from HUMAN and the array is closed up.

If it is the computer's turn to play (**IF NOT** WHO), a call to COMPLAY determines the computer's playing strategy.

**Function COMPLAY** begins, just as HUMPLAY did, by testing the first card in the hand to determine if any card is playable. If the computer can play, then COMPLAY simply sends each playable card in turn to COUNTSCORE to find out how many points it would yield if played:

```
POINTS := COUNTSCORE(COMP[INDEX]);
```

Then the work of COUNTSCORE must be carefully undone, pending a decision on the computer's play:

```
CNTNUM := CNTNUM - 1;
CURCOUNT := CURCOUNT - TMP;
```

Finally, the index of the best scoring card in COMP is returned to COUNT.

**COUNT (continued).** The computer then displays its card (call to SHOWCARD), counts up the points that the play is worth (call to COUNTSCORE) and "pegs" the points (call to ADDPOINTS). The computer's hand is reduced by one, and the current count is displayed.

The play continues until the last card is played; a CASE statement (**CASE** LAST **OF** . . . ) awards a point to the player of the last card. (The value of WHO is reassigned so that the other player plays next.) Notice that this point is pegged both at the end of a series (i.e., when neither player can play):

```
IF (HUMCANT AND COMPCANT)
```

and at the end of a round (i.e., when all cards have been played):

```
OR ((HUMLEFT = 0) AND (COMLEFT = 0))
```

At the end of COUNT, the human's original hand is put back into the array HUMAN, anticipating the "showing" that is about to take place:

```
FOR CURCOUNT := 1 TO PLAYSIZE
  HUMAN[CURCOUNT] := OLDHUMAN[CURCOUNT]
```

**PLAY (continued).** Now comes the nasty little move that the computer has been looking forward to since the very beginning of the round. The player's hand is displayed (five calls to SHOWCARD—four for the player's hand, and one for the starter) and procedure CHECK is called to find out if the human can add. Notice that CHECK is sent two values:

```
CHECK(HPOINTS, 'HOW MANY POINTS IN YOUR HAND?')
```

HPOINTS is, of course, the already-computed true point value of the player's hand.

**Procedure CHECK** displays the prompt it was passed, reads the player's input, and compares it to the true score (stored in the formal parameter NUM). If a discrepancy is found, the computer pegs the difference. Notice the use of the ABS function to find the absolute value of the difference:

```
ADDPOINTS(FALSE, ABS(NUM - USER));
```

Another call to ADDPOINTS pegs either the player's points or the player's incorrect calculation, whichever is smaller.

**PLAY (continued).** There are two different blocks for the "showing" of the player's cards. If the crib belongs to the computer, the first block is executed. If the crib belongs to the player, the second block contains a second call to CHECK for the player's estimate of the points in the crib:

```
CHECK(CRBPNTS, 'HOW MANY POINTS IN YOUR CRIB? ');
```

```
 (*$S+
 * CRIBBAGE
 *)
PROGRAM CRIBBAGE(INPUT,OUTPUT);
USES APPLESTUFF;
CONST
   DECKSIZE     = 52;         (* NUMBER OF CARDS IN THE DECK *)
   DEALSIZE     =  6;         (* NUMBER DEALT TO A PLAYER *)
   SCRIBSIZE    =  6;         (* THE SIZE OF THE CRIB (ACTUALLY 2 LARGER) *)
   PLAYSIZE     =  4;         (* NUMBER OF CARDS YOU PLAY WITH *)
   RANKSIZE     = 13;         (* CARDS ARE RANKED FROM 1 -> 13 *)
   WINPOINTS    =121;         (* WHAT IT TAKES TO WIN *)

TYPE
   SUITYPE      = (HEARTS,DIAMONDS,CLUBS,SPADES);
   RANKTYPE     = 0..RANKSIZE;

   CARD  = RECORD
      RANK : RANKTYPE;        (* 0 IMPLIES NIL CARD *)
      SUIT : SUITYPE          (* SUIT OF A CARD *)
   END;

   HANDTYPE     = ARRAY[1..DEALSIZE] OF CARD;

VAR
   DECK  : ARRAY[1..DECKSIZE] OF CARD;(* THE ENTIRE DECK *)
   COMP  : HANDTYPE;                  (* THE COMPUTER'S HAND *)
   HUMAN : HANDTYPE;                  (* THE HUMAN'S HAND *)
   CRIB  : HANDTYPE;                  (* THE CRIB *)
   COMMON: CARD;                      (* CARD TURNED OVER FROM DECK *)
   HSCORE: INTEGER;                   (* NUMBER OF POINTS HUMAN HAS *)
   CSCORE: INTEGER;                   (* NUMBER OF POINTS COMPUTER HAS *)

   (*
    * ADDPOINTS
    * ADDS AMOUNT OF POINTS TO PLAYER'S SCORE
    *)
   PROCEDURE ADDPOINTS(WHO : BOOLEAN;AMOUNT:INTEGER);
      FORWARD;

   (*$I#5:CARD.TEXT*)

   (*
    * GETELEMENT
    * GETS A CARD FROM THE USER THAT
    * CORRESPONDS TO ONLY ONE CARD
    * IN HIS HAND
    *)
   FUNCTION GETELEMENT:INTEGER;
   VAR
      IRANK,            (* RANK INPUT BY USER *)
      ISUIT: CHAR;      (* SUIT INPUT BY USER *)

      RANK : RANKTYPE;  (* RANK FROM USER *)
      SUIT : SUITYPE;   (* SUIT FROM USER *)
   WHICH: INTEGER;   (* WHICH NUMBER TO RETURN *)
   INDEX:1..DEALSIZE;(* USED TO GO THROUGH THE HAND *)
   MANY :-5..4;       (* HOW MANY CARDS CORRESPOND *)
BEGIN
   READ(IRANK);
   IF NOT EOLN THEN
      READLN(ISUIT)
   ELSE
      READLN;

   IF NOT (IRANK IN ['A','2'..'9','T','J','Q','K']) OR
      NOT (ISUIT IN ['S','D','C','H',' ']) THEN
   BEGIN
      WRITELN('ILLEGAL INPUT');
      WRITE('WHICH CARD? ');
      GETELEMENT := GETELEMENT
   END
   ELSE
   BEGIN
      CASE IRANK OF
         'A' : RANK := 1;
         '2','3','4','5','6','7','8','9':
               RANK := ORD(IRANK) - ORD('0');
         'T' : RANK := 10;
         'J' : RANK := 11;
         'Q' : RANK := 12;
         'K' : RANK := 13
      END;

      CASE ISUIT OF
         'S' : SUIT := SPADES;
         'H' : SUIT := HEARTS;
         'D' : SUIT := DIAMONDS;
         'C' : SUIT := CLUBS
      END;

      MANY := 0;
      WHICH := 0;
      FOR INDEX := 1 TO DEALSIZE DO
      BEGIN
         IF HUMAN[INDEX].RANK = RANK THEN
         BEGIN
            MANY := MANY +1;
            IF MANY > 0 THEN
               WHICH := INDEX;
            IF ISUIT <> ' ' THEN
               IF HUMAN[INDEX].SUIT = SUIT THEN
                  MANY := -5
         END
      END;
      IF MANY = 0 THEN
      BEGIN
         WRITELN('THERE IS NO SUCH CARD!!');
         WRITE('WHICH CARD? ');
         GETELEMENT := GETELEMENT
      END;
      IF MANY > 1 THEN
      BEGIN
         WRITELN('THERE IS MORE THAN ONE ',IRANK);
         WRITE('PLEASE BE MORE SPECIFIC  ');
         GETELEMENT := GETELEMENT
      END;

      IF (MANY = 1) OR (MANY < 0 ) THEN
         GETELEMENT := WHICH
   END
END; (* GETELEMENT *)



(*
 * TOCRIB
 * GET THE HUMAN TO PUT 2 CARDS IN THE
 * CRIB
 *)
PROCEDURE TOCRIB;
VAR
   CARDNUM : 1..DEALSIZE;  (* INDEX INTO HAND *)
   NUMGONE:  0..1;         (* NUMBER OF CARDS REMOVED *)
BEGIN

   FOR NUMGONE := 0 TO 1 DO
   BEGIN
      (* PRINT OUT CARDS *)
      WRITE('REMOVE WHICH CARD? [');
      FOR CARDNUM := 1 TO (DEALSIZE - NUMGONE) DO
         SHOWCARD(HUMAN[CARDNUM]);
      WRITE('] ');
      CARDNUM := GETELEMENT;
      CRIB[NUMGONE+1] := HUMAN[CARDNUM];
      WHILE CARDNUM <= (DEALSIZE - 1) DO
      BEGIN
         HUMAN[CARDNUM] := HUMAN[CARDNUM+1];
         CARDNUM := CARDNUM +1
      END;
      HUMAN[CARDNUM].RANK := 0;
   END
END; (* TOCRIB *)

(*
 * SORT
 * SORT THE FIRST 'N' ELEMENTS IN
 * THE ARRAY 'HAND' USING A BUBBLE
 * SORT
 *)
PROCEDURE SORT(N:INTEGER;VAR HAND : HANDTYPE);
VAR
   TOUCHED : BOOLEAN;          (* WHETHER ANYTHING WAS MOVED *)
   INDEX   : 1..DEALSIZE;      (* INDEX INTO HAND ARRAY *)
   TMP     : CARD;             (* TEMPORARY *)
```

```
BEGIN
   REPEAT
      TOUCHED := FALSE;
      FOR INDEX := 1 TO (N -1) DO
         IF HAND[INDEX].RANK > HAND[INDEX+1].RANK THEN
         BEGIN
            TMP := HAND[INDEX];
            HAND[INDEX] := HAND[INDEX+1];
            HAND[INDEX+1] := TMP;
            TOUCHED := TRUE
         END
   UNTIL NOT TOUCHED
END; (* SORT *)

(*$I#5:SCORE.TEXT*)


(*
 * COMPCRIB
 * HAVE THE COMPUTER DECIDE WHAT TO
 * PUT IN THE CRIB (ALL IT DOES IS DECIDE
 * WHAT GROUP OF 4 CARDS GIVES IT THE
 * MOST POINTS)
 *)
PROCEDURE COMPCRIB;
TYPE
   BESTREC = RECORD
      POINTS : INTEGER;
      FIRST,
      SECOND : 1..DEALSIZE
   END;
VAR
   TMP      : HANDTYPE;
   BEST     : BESTREC;
   POINTS   : INTEGER;        (* NUMBER OF POINTS A HAND WOULD SCORE *)
   I,J      : INTEGER;        (* INDEXES *)

   (*
    * COMPSCORE
    * REMOVES THE TWO PASSED CARDS,
    * THEN RETURNS THE NUMBER OF POINTS
    * THE RESULTING HAND HAS
    *)
   FUNCTION COMPSCORE:INTEGER;
   VAR
      INDEX : INTEGER;
      POINTS : INTEGER;
      NUM    : 1..DEALSIZE;
   BEGIN
      NUM := 1;
      FOR INDEX := 1 TO (I-1) DO
      BEGIN
         TMP[NUM] := COMP[INDEX];
         NUM := NUM +1
      END;
      FOR INDEX := (I+1) TO (J-1) DO
      BEGIN
         TMP[NUM] := COMP[INDEX];
         NUM := NUM + 1
      END;

      FOR INDEX := (J+1) TO DEALSIZE DO
      BEGIN
         TMP[NUM] := COMP[INDEX];
         NUM := NUM +1
      END;


      TMP[5].RANK := 0;

      COMPSCORE := SCORE(TMP);
   END; (* COMPSCORE *)


BEGIN (* COMPCRIB *)
   BEST.POINTS := -1;        (* EVEN 0 IS BETTER *)

   SORT(6,COMP);

   FOR I := 1 TO (DEALSIZE -1) DO
      FOR J := I+1 TO DEALSIZE DO
      BEGIN
         POINTS := COMPSCORE;
         IF POINTS > BEST.POINTS THEN
         BEGIN
            BEST.POINTS := POINTS;
            BEST.FIRST := I;
            BEST.SECOND := J
         END
      END;
   J := 1;
   FOR I := 1 TO (BEST.FIRST -1) DO
   BEGIN
      TMP[J] := COMP[I];
      J := J + 1
   END;

   FOR I := (BEST.FIRST +1) TO (BEST.SECOND -1) DO
   BEGIN
      TMP[J] := COMP[I];
      J := J + 1
   END;

   FOR I := (BEST.SECOND+1) TO DEALSIZE DO
   BEGIN
      TMP[J] := COMP[I];
      J := J + 1
   END;
   CRIB[3] := COMP[BEST.FIRST];
   CRIB[4] := COMP[BEST.SECOND];

   FOR I := 1 TO PLAYSIZE DO
      COMP[I] := TMP[I];


END; (* COMPCRIB *)


(*
 * COUNT
 * CONTROLS THE COUNTING
 *)
PROCEDURE COUNT(WHO:BOOLEAN);
VAR
   OLDHUMAN    : ARRAY[1..4] OF CARD;  (* HOLD THE OLD CARDS *)
   CURCOUNT    : INTEGER;             (* CURRENT VALUE OF THE COUNT *)
   HUMCANT,                           (* THE HUMAN CAN'T MAKE A PLAY *)
   COMPCANT    : BOOLEAN;             (* THE COMPUTER CAN'T PLAY *)
   CNTHAND     : ARRAY[1..8] OF CARD; (* WHEN A CARD IS PLAYED,IT IS PUT HERE*)
   LAST        : 0..2;                (* USED TO TELL WHO PLAYED LAST *)
   CNTNUM      : 1..8;                (* INDEX INTO CNTHAND *)
   LASTCNT     : INTEGER;             (* LAST VALUE OF CURCOUNT *)

   HUMLEFT     : 0..PLAYSIZE;         (* HOW MANY CARDS THE HUMAN HAS LEFT *)
   COMLEFT     : 0..PLAYSIZE;         (*  "   "    "    "  COMPUTER ... *)
   I           : -1..PLAYSIZE;        (* POSSIBLE CARDS TO PLAY *)
   POINTS      : INTEGER;             (* POINTS FOR A PLAY *)


   (*
    * COUNTSCORE
    * SCORE THE PLAY HAND, NOTE THAT I
    * CAN'T USE THE 'SCORE' FUNCTION,BECAUSE I
    * AM ONLY INTERESTED IN THE TOP CARD, NOT
    * THE WHOLE STACK
    *)
   FUNCTION COUNTSCORE(NEWCARD : CARD):INTEGER;
   VAR
      RETURN  :  INTEGER;             (* AMOUNT TO RETURN *)
      MATCHED :  0..8;                (* NUMBER OF CARDS MATCHED *)
      INDEX   :  0..8;                (* INDEX INTO CNTHAND *)
   BEGIN
      RETURN := 0;
      MATCHED := 0;
      CNTHAND[CNTNUM] := NEWCARD;
      IF CNTHAND[CNTNUM].RANK > 10 THEN
         CURCOUNT := CURCOUNT + 10
      ELSE
         CURCOUNT := CURCOUNT + CNTHAND[CNTNUM].RANK;
```

```
IF CNTNUM = 1 THEN
BEGIN
   CNTNUM := CNTNUM +1;
   COUNTSCORE := 0
END
ELSE
BEGIN

   IF (CURCOUNT = 15) OR (CURCOUNT = 31) THEN
      RETURN := 2;
   INDEX := CNTNUM;

   (*
    * FIRST FIND ALL PAIRS
    * STARTING FROM THE BACK
    *)

   WHILE INDEX >= 2 DO
   BEGIN
      IF CNTHAND[INDEX].RANK = CNTHAND[INDEX-1].RANK THEN
         MATCHED := MATCHED +1
      ELSE
         INDEX := 1;
      INDEX := INDEX -1
   END;

   CASE MATCHED OF
      0 : ;
      1 : RETURN := RETURN + 2;
      2 : RETURN := RETURN + 6;
      3 : RETURN := RETURN + 12
   END;

   MATCHED := 0;
   (*
    * NOW LOOK FOR RUNS
    *)
   INDEX := CNTNUM;
   WHILE INDEX >= 2 DO
   BEGIN
      IF CNTHAND[INDEX].RANK = (CNTHAND[INDEX-1].RANK -1) THEN
         MATCHED := MATCHED +1
      ELSE
         INDEX := 1;
      INDEX := INDEX -1
   END;

      CNTNUM := CNTNUM +1;

      IF MATCHED > 2 THEN
         RETURN := RETURN + MATCHED +1;

      COUNTSCORE := RETURN
   END
END; (* COUNTSCORE *)
```

```
(*
 * HUMPLAY
 * GET THE HUMAN TO PLAY A CARD.
 * IF ONLY ONE CARD IS PLAYABLE,
 * PLAY IT FOR HIM
 *)
FUNCTION HUMPLAY:INTEGER;
VAR
   I,J : INTEGER;          (* INDICES *)
BEGIN

IF HUMAN[1].RANK > 10 THEN     (* A JACK (11) IS ONLY WORTH 10 *)
   I := 10
ELSE
   I := HUMAN[1].RANK;
(*
 * TEST TO SEE IF THE HUMAN
 * CAN MAKE ANY PLAY
 *)
IF ( HUMLEFT <= 0 ) OR ( (I + CURCOUNT) > 31 ) THEN
BEGIN
   HUMCANT := TRUE;
   HUMPLAY := -1               (* RETURN AN IMPOSSIBILITY *)
END
ELSE
BEGIN
   LAST := 1;
   HUMCANT := FALSE;
   IF HUMAN[2].RANK > 10 THEN
      I := 10
   ELSE
      I := HUMAN[2].RANK;
   (*
    * CHECK TO SEE IF THE ONLY
    * PLAYABLE CARD IS THE FIRST ONE
    *)
   IF (HUMLEFT = 1) OR ((CURCOUNT+I) > 31) THEN
      HUMPLAY := 1
   ELSE
   BEGIN
      J := 0;
      WHILE J = 0 DO
      BEGIN
         WRITE('PLAY WHICH? [');
         FOR I := 1 TO PLAYSIZE DO
            IF HUMAN[I].RANK <> 0 THEN
               SHOWCARD(HUMAN[I]);
               WRITE('] ');
            I := GETELEMENT;
            IF HUMAN[I].RANK > 10 THEN
               J := 10
            ELSE
```

```
               J := HUMAN[I].RANK;
            IF (J + CURCOUNT) > 31 THEN
            BEGIN
               WRITELN('SORRY, THAT IS MORE THAN 31');
               J := 0
            END;
         END;
         HUMPLAY := I
      END
   END
END; (* HUMPLAY *)

(*
 * COMPLAY
 * HAVE THE COMPUTER PLAY A CARD.
 * ALL IT DOES IS CHECK THROUGH ALL
 * ITS CARDS, THEN PLAY THE ONE THAT GIVES THE
 * MOST POINTS. IF NONE GIVE POINTS,
 * THEN PLAY THE FIRST CARD
 *)

FUNCTION COMPLAY:INTEGER;
VAR
   INDEX  : 1..PLAYSIZE;
   POINTS : INTEGER;       (* POINTS FOR A CARD *)
   BEST   : INTEGER;       (* BEST POINTS FOR A CARD (SO FAR) *)

   TMP    : 0..10;         (* TEMPORARY FOR MY OWN SANITY *)
   RETURN : 1..PLAYSIZE;   (* NUMBER TO RETURN *)
BEGIN
   BEST := -1;             (* ANYTHING IS BETTER THAN THIS *)
   IF COMP[1].RANK > 10 THEN
      TMP := 10
   ELSE
      TMP := COMP[1].RANK;

   IF (COMLEFT = 0) OR ((TMP + CURCOUNT) > 31) THEN
   BEGIN
      COMPCANT := TRUE;
      COMPLAY := -1        (* AN IMPOSSIBILITY *)
   END
   ELSE
   BEGIN
      COMPCANT := FALSE;
      LAST := 2;

      FOR INDEX := 1 TO COMLEFT DO
      BEGIN
         IF COMP[INDEX].RANK > 10 THEN
            TMP := 10
         ELSE
            TMP := COMP[INDEX].RANK;
```

```
             IF (TMP <> 0 ) AND ( (TMP + CURCOUNT) <= 31) THEN
             BEGIN
                POINTS := COUNTSCORE(COMP[INDEX]);
                CNTNUM := CNTNUM -1;            (* UNDO WHAT COUNTSCORE DID *)
                CURCOUNT := CURCOUNT - TMP;
                IF POINTS > BEST THEN
                BEGIN
                   BEST := POINTS;
                   RETURN := INDEX
                END
             END
          END;
          COMPLAY := RETURN
       END
    END; (* COMPLAY *)
 BEGIN (* COUNT *)
    HUMLEFT := PLAYSIZE;
    COMLEFT := PLAYSIZE;
    HUMCANT := FALSE;
    COMPCANT := FALSE;
    LAST := 0;
    CNTNUM := 1;

    IF COMMON.RANK = 11 THEN
    BEGIN
       IF WHO THEN
       BEGIN
          WRITELN('I GET A POINT FOR HIS NIBBS');
          ADDPOINTS(FALSE,1)
       END
       ELSE
       BEGIN
          WRITELN('YOU GET A POINT FOR HIS NIBBS');
          ADDPOINTS(TRUE,1)
       END
    END;

    FOR CURCOUNT := 1 TO PLAYSIZE DO
       OLDHUMAN[CURCOUNT] := HUMAN[CURCOUNT];
    CURCOUNT := 0;
(*
 * WHILE SOMEONE CAN STILL PLAY
 * A CARD, CONTINUE
 *)
 WHILE ( HUMLEFT > 0 ) OR ( COMLEFT > 0 ) DO
 BEGIN
    IF WHO THEN
    BEGIN
       WHO := FALSE;
       I := HUMPLAY;
       IF I > 0 THEN
       BEGIN
          WRITE('YOU PLAYED A ');
          SHOWCARD(HUMAN[I]);
          WRITELN;
```

```
      POINTS := COUNTSCORE(HUMAN[I]);
      IF POINTS > 0 THEN
      BEGIN
         WRITELN('YOU GOT ',POINTS);
         ADDPOINTS(TRUE,POINTS)
      END;
      WHILE I <= (PLAYSIZE -1) DO
      BEGIN
         HUMAN[I] := HUMAN[I+1];
         I := I + 1
      END;
      HUMAN[HUMLEFT].RANK := 0;
      HUMLEFT := HUMLEFT -1;
   END
END;
IF NOT WHO THEN
BEGIN
   WHO := TRUE;
   I := COMPLAY;
   IF I > 0 THEN
   BEGIN
      WRITE('I PLAY A ');
      SHOWCARD(COMP[I]);
      WRITELN;
      POINTS := COUNTSCORE(COMP[I]);
      IF POINTS > 0 THEN
      BEGIN
         WRITELN('I GOT ',POINTS);
         ADDPOINTS(FALSE,POINTS)
      END;
      WHILE I <= (PLAYSIZE -1) DO
      BEGIN
         COMP[I] := COMP[I+1];
         I := I + 1
      END;
      COMP[COMLEFT].RANK := 0;
      COMLEFT := COMLEFT -1;
   END
END;
IF LASTCNT <> CURCOUNT THEN
   WRITELN('TOTAL IS ',CURCOUNT);
LASTCNT := CURCOUNT;
(*
 * TEST TO SEE IF NO MORE CARDS
 * CAN BE PLAYED
 *)
IF  (HUMCANT AND COMPCANT) OR ( (HUMLEFT = 0) AND (COMLEFT = 0)) THEN
BEGIN
     CASE LAST OF
        0 : ;
        1 : (* THE HUMAN MADE THE LAST PLAY *)
           BEGIN
              WRITELN('YOU GOT A POINT FOR LAST CARD');
              ADDPOINTS(TRUE,1);
              WHO := FALSE
           END;
```

```
            2 : (* THE COMPUTER WENT LAST *)
               BEGIN
                  WRITELN('I GOT A POINT FOR LAST CARD');
                  ADDPOINTS(FALSE,1);
                  WHO := TRUE
               END
         END;
         WRITELN;
         WRITELN('TOTAL IS NOW 0');
         WRITELN;
         HUMCANT := FALSE;
         COMPCANT := FALSE;
         LAST := 0;
         CNTNUM := 1;
         CURCOUNT := 0;
      END;

   END;
   FOR CURCOUNT := 1 TO PLAYSIZE DO
      HUMAN[CURCOUNT] := OLDHUMAN[CURCOUNT]
END; (* COUNT *)

PROCEDURE PLAY(WHO:BOOLEAN);
VAR
   CPOINTS,          (* POINTS IN COMPUTER'S HAND *)
   HPOINTS,          (* HUMAN'S *)
   CRBPNTS: INTEGER; (* IN CRIB *)
   USER  :  INTEGER; (* INPUT FROM USER *)

   (*
    * CHECK
    * GETS A NUMBER FROM THE USER, AND
    * IF IT DIFFERS FROM THE NUMBER
    * PASSED, THEN THE COMPUTER SCORES THE DIFFERENCE
    *)
   PROCEDURE CHECK(NUM : INTEGER; QUESTION : STRING);
   BEGIN
      WRITE(QUESTION);
      READLN(USER);
      IF USER <> NUM THEN
      BEGIN
         WRITELN('YOU SHOULD HAVE TAKEN ',NUM,' POINTS');
         WRITELN('I GET ',ABS(NUM-USER));
         ADDPOINTS(FALSE,ABS(NUM-USER));
         IF USER > NUM THEN
            USER := NUM
      END;
      IF USER > 0 THEN
         ADDPOINTS(TRUE,USER);
      WRITELN
   END; (* CHECK *)

BEGIN (* PLAY *)
   WHILE TRUE DO
   BEGIN
      SHUFFLE;
```

```
DEAL;
IF WHO THEN
   WRITELN('IT''S MY CRIB')
ELSE
   WRITELN('IT''S YOUR CRIB');
SORT(6,HUMAN);
TOCRIB;
COMPCRIB;
SORT(4,CRIB);


IF WHO THEN
BEGIN
   REPEAT
      WRITE('CUT WHICH CARD [1-40]? ');
      READLN(USER)
   UNTIL USER IN [1..40];
   COMMON := DECK[USER+12]
END
ELSE
BEGIN
   COMMON := DECK[13+RANDOM MOD 40]
END;
WRITE('THE COMMON CARD IS ');
SHOWCARD(COMMON);
WRITELN;
CPOINTS := SCORE(COMP);
HPOINTS := SCORE(HUMAN);
CRBPNTS := SCORE(CRIB);
WRITELN;
COUNT(WHO);

WRITELN;
IF WHO THEN
BEGIN
   WRITE('[ ');
   FOR USER := 1 TO PLAYSIZE DO
      SHOWCARD(HUMAN[USER]);
   WRITE('] [ ');
   SHOWCARD(COMMON);
   WRITELN(']');
   CHECK(HPOINTS,'HOW MANY POINTS IN YOUR HAND? ');
   WRITELN('I HAVE ',CPOINTS,' POINTS IN MY HAND');
   ADDPOINTS(FALSE,CPOINTS);
   WRITELN('I HAVE ',CRBPNTS,' POINTS IN MY CRIB');
   ADDPOINTS(FALSE,CRBPNTS)
END
ELSE
BEGIN
   WRITELN('I HAVE ',CPOINTS,' POINTS IN MY HAND');
   ADDPOINTS(FALSE,CPOINTS);
   WRITELN;
   WRITE('[ ');
   FOR USER := 1 TO PLAYSIZE DO
      SHOWCARD(HUMAN[USER]);
   WRITE('] [ ');
```

```
         SHOWCARD(COMMON);
         WRITELN(']');
         CHECK(HPOINTS,'HOW MANY POINTS IN YOUR HAND? ');
         WRITELN;
         WRITE('[ ');
         FOR USER := 1 TO PLAYSIZE DO
            SHOWCARD(CRIB[USER]);
         WRITE('] [ ');
         SHOWCARD(COMMON);
         WRITELN(']');
         CHECK(CRBPNTS,'HOW MANY POINTS IN YOUR CRIB? ');
      END;
      WHO := NOT WHO;
   END;
END; (* PLAY *)
  (*
   * ADDPOINTS
   * NOTE THAT THIS USES THE UCSD
   * INTRINSIC 'EXIT'.
   *)
  PROCEDURE ADDPOINTS;
  BEGIN
     IF WHO THEN
     BEGIN
        HSCORE := HSCORE + AMOUNT;
        WRITELN('YOUR SCORE IS NOW ',HSCORE,' POINTS');
        IF HSCORE >= WINPOINTS THEN
           EXIT(PLAY)
     END
     ELSE
     BEGIN
        CSCORE := CSCORE + AMOUNT;
        WRITELN('MY SCORE IS NOW ',CSCORE,' POINTS');
        IF CSCORE >= WINPOINTS THEN
           EXIT(PLAY)
     END
  END;(* ADDPOINTS *)
BEGIN
   (*
    * THIS I/O IS REALLY JUST SO
    * THAT RANDOMIZE WILL PROPERLY
    * SEED THE RANDOM NUMBER GENERATOR
    *)
   WRITELN('*****  WELCOME TO CRIBBAGE  *****');
   RANDOMIZE;
   MAKEDECK;
   HSCORE := 0;
   CSCORE := 0;
   PLAY(((RANDOM MOD 2) = 0));

   IF CSCORE > HSCORE THEN
      WRITELN('THE COMPUTER HAS WON THIS GAME')
   ELSE
      WRITELN('CONGRATULATIONS, YOU HAVE BEATEN THE COMPUTER')

END. (* CRIBBAGE 7/4/81 *)
```

## CARD.TEXT file

```
(*
 * MAKEDECK
 * CREATES THE DECK.
 * CALLED ONCE AT THE BEGINNING
 *)
PROCEDURE MAKEDECK;
VAR
   CARDNUM : 0..DECKSIZE;
   RANK    : RANKTYPE;
   SUIT    : SUITTYPE;
BEGIN
   CARDNUM := 0;

   FOR SUIT := HEARTS TO SPADES DO
      FOR RANK := 1 TO RANKSIZE DO
      BEGIN
         CARDNUM := CARDNUM +1;
         DECK[CARDNUM].RANK := RANK;
         DECK[CARDNUM].SUIT := SUIT;
      END
END; (* MAKEDECK *)

(*
 * SHUFFLE
 * GO LINEARLY THROUGH THE DECK
 * EXCHANGING CARDS
 *)
PROCEDURE SHUFFLE;
VAR
   CURCARD : 1..DECKSIZE;     (* CURRENT CARD NUMBER *)
   NEWCARD : 1..DECKSIZE;     (* WHERE TO PUT THE CURRENT CARD *)
   TMP     : CARD;            (* TEMPORARY HOLDING VARIABLE *)
BEGIN
   (*
    * FOR EACH CARD IN THE DECK
    *)
   FOR CURCARD := 1 TO DECKSIZE DO
   BEGIN
      (*
       * RANDOMLY CHOOSE A DIFFRENT CARD
       *)
      NEWCARD := 1 + RANDOM MOD DECKSIZE;
      (*
       * EXCHANGE CURCARD WITH
       * NEWCARD
       *)
      TMP := DECK[NEWCARD];
      DECK[NEWCARD] := DECK[CURCARD];
      DECK[CURCARD] := TMP
   END
END; (* SHUFFLE *)
```

```
(*
 * DEAL
 * GIVE THE FIRST 6 CARDS TO THE HUMAN,
 * THE NEXT 6 TO THE COMPUTER
 *)
PROCEDURE DEAL;
VAR
   I : 1..DEALSIZE;                          (* INDEX *)
BEGIN
   COMMON.RANK := 0;
   FOR I := 1 TO DEALSIZE DO
   BEGIN
      HUMAN[I] := DECK[I];
      COMP[I] := DECK[I+DEALSIZE]
   END
END; (* DEAL *)


(*
 * SHOWCARD
 * PRINTS OUT A CARD IN A SEMIREADABLE
 * FORMAT.
 *)
PROCEDURE SHOWCARD(TOSHOW : CARD);
BEGIN
   CASE TOSHOW.RANK OF

      1 : WRITE('A');
      2,3,4,5,6,7,8,9: WRITE(TOSHOW.RANK:1);
      10: WRITE('T');
      11: WRITE('J');
      12: WRITE('Q');
      13: WRITE('K')
   END;

   CASE TOSHOW.SUIT OF
      HEARTS   : WRITE('H');
      CLUBS    : WRITE('C');
      DIAMONDS: WRITE('D');
      SPADES   : WRITE('S')
   END;
   WRITE(' ')
END; (* SHOWCARD *)
```

## SCORE.TEXT file

```
(*
 * SCORE
 * FIGURE OUT HOW MANY POINTS IN
 * A HAND
 *)
FUNCTION SCORE(HAND : HANDTYPE):INTEGER;
CONST
   JACK  =  11;      (* RANK VALUE OF A JACK *)
VAR
   INDEX : 1..5;
   RETURN : INTEGER;
   CARDMAX: 4..5;            (* FIVE MEANS WE ARE COUNTING
                                THE COMMON CARD *)

(*
 * FIFTEEN
 * COUNT UP HOW MANY WAYS TO MAKE
 * FIFTEEN THERE ARE, AND RETURN
 * THAT NUMBER * 2
 *)
FUNCTION FIFTEEN(HAND:HANDTYPE):INTEGER;
VAR
   WAYS    : INTEGER;        (* NUMBER OF WAYS TO MAKE FIFTEEN *)
   I,J,K   : 1..DEALSIZE;    (* INDICES *)
   SUM     : INTEGER;        (* SUM OF ALL THE CARDS *)
BEGIN


   SUM := 0;

   (*
    * FIGURE OUT SUM OF CARDS, AND
    * SET FACE CARDS TO BE WORTH 10
    * EACH
    *)
   FOR I := 1 TO CARDMAX DO
   BEGIN
      IF HAND[I].RANK > 10 THEN
         HAND[I].RANK := 10;
      SUM := SUM + HAND[I].RANK
   END;

   WAYS := 0;
   IF SUM = 15 THEN
      WAYS := 1;

   FOR I := 1 TO CARDMAX DO
   BEGIN
      WITH HAND[I] DO
      BEGIN
         (*
          * NOTE THAT I TEST FOR
          * < SO THAT AN UNNEEDED
          * FOR LOOP WON'T BE EXECUTED
          *)

         IF (SUM - RANK) <= 15 THEN
         BEGIN
            IF (SUM - RANK) = 15 THEN
               WAYS := WAYS + 1
         END
         ELSE
            FOR J := I+1 TO CARDMAX DO
            BEGIN


              (*
               * KEEP SUBTRACTING CARDS
               * OFF THE SUM AND SEE IF
               * WE GET 15; IF SO THEN
               * THIS IS A WAY OF GETTING
               * 15
               *)
              IF (SUM - RANK - HAND[J].RANK) <= 15 THEN
              BEGIN
                 IF (SUM - RANK - HAND[J].RANK) = 15 THEN
                    WAYS := WAYS +1
              END
              ELSE
                 FOR K := J+1 TO CARDMAX DO
                    IF (SUM - RANK - HAND[J].RANK - HAND[K].RANK) = 15 THEN
                       WAYS := WAYS +1
           END
     END
  END;

  FIFTEEN := WAYS * 2
END; (* FIFTEEN *)


(*
 * RUN
 * FIGURE OUT THE NUMBER OF POINTS
 * IN A HAND DUE TO A RUN
 *)
FUNCTION RUN(HAND:HANDTYPE):INTEGER;
VAR
   MULT    :  INTEGER;        (* MULT FACTOR (3445 HAS 2*3 POINTS) *)
   MATCHED:  0..5;            (* NUMBER OF MATCHED CARDS (1235, MATCHES 2 *)
   I       :  1..DEALSIZE;    (* INDEX INTO HAND *)
   RETURN  :  INTEGER;        (* POINTS TO RETURN *)
   SEEN    :  BOOLEAN;        (* TRUE IF WE HAVE SEEN A PAIR *)
```

```
BEGIN
   MULT := 1;
   MATCHED := 0;
   RETURN := 0;
   SEEN := FALSE;

   FOR I := 1 TO (CARDMAX -1) DO
   BEGIN
      (*
       * 2 3 3 4 IS TWO THREE-
       * CARD RUNS. CHECK FOR
       * PAIRS INSIDE RUNS (LIKE
       * THE TWO THREE'S )
       *)
      IF HAND[I].RANK = HAND[I+1].RANK THEN
      BEGIN
         IF SEEN THEN
            MULT := MULT + 1
         ELSE
            MULT := MULT + 2;

         SEEN := TRUE;
         (*
          * TEST FOR THE LAST TWO
          * CARDS BEING THE SAME
          * (EX. 2 7 8 9 9).
          *)
         IF I = (CARDMAX -1) THEN
         BEGIN
            IF MULT >= 2 THEN
               MULT := MULT -1;
            IF MATCHED >= 2 THEN
               RETURN := RETURN + ((MATCHED+1)*MULT);
            MATCHED := 0;
         END
      END
      ELSE
      BEGIN
         (*
          * CONTINUATION OF A RUN
          *)
         SEEN := FALSE;
         IF HAND[I].RANK = (HAND[I+1].RANK -1) THEN
            MATCHED := MATCHED +1
         ELSE
         BEGIN
            IF MULT >= 2 THEN
               MULT := MULT -1;
            IF MATCHED >= 2 THEN
               RETURN := RETURN + ((MATCHED+1)*MULT);
            MULT := 1;      (* END OF A RUN, RESET *)
            MATCHED := 0
         END
      END
   END;
   IF MULT >= 2 THEN
      MULT := MULT -1;
   IF MATCHED >= 2 THEN
      RETURN := RETURN + ((MATCHED+1)*MULT);

   RUN := RETURN

END; (* RUN *)


(*
 * PAIR
 * FIND OUT HOW MANY
 * PAIRS IN A HAND.
 *)
FUNCTION PAIR(HAND:HANDTYPE):INTEGER;
VAR
  RETURN : INTEGER;
  MATCHED: 0..4;
  INDEX  : 1..DEALSIZE;
BEGIN

   RETURN := 0;
   MATCHED := 0;

   FOR INDEX := 1 TO (CARDMAX -1) DO
   BEGIN
      IF HAND[INDEX].RANK = HAND[INDEX+1].RANK THEN
         MATCHED := MATCHED +1;

      (*
       * I CAN'T USE AN ELSE, BECAUSE
       * OF THE NEED TO CHECK FOR PAIRS
       * AT THE END OF THE CARDS
       *)

      IF ( HAND[INDEX].RANK <> HAND[INDEX+1].RANK)OR(INDEX=CARDMAX -1) THEN
      BEGIN
         CASE MATCHED OF
            0 : ;
            1 : RETURN := RETURN + 2;
            2 : RETURN := RETURN + 6;
            3 : RETURN := RETURN + 12
         END;
         MATCHED := 0
      END
   END;

   PAIR := RETURN
END; (* PAIR *)

(*
 * FLUSH
 * SEE IF THERE IS A FLUSH IN THE ORIGINAL
 * HAND OF FOUR CARDS; THEN CHECK THE COMMON CARD.
 *)
```

```
FUNCTION FLUSH(HAND:HANDTYPE):INTEGER;
VAR
   RETURN  : INTEGER;          (* HOW MUCH HAND'S WORTH *)
   INDEX   : 1..DEALSIZE;      (* INDEX INTO HAND ARRAY *)
BEGIN

   RETURN := 1;                (* INITIALIZE RETURN TO 1 AND THEN
                                  USE THE FOR LOOP TO MAKE THE 3
                                  NECESSARY COMPARISONS TO SEE IF
                                  THE ORIGINAL HAND IS A FLUSH. *)

   FOR INDEX := 1 TO 3 DO
      IF HAND[INDEX].SUIT = HAND[INDEX+1].SUIT THEN
         RETURN := RETURN +1;

   (* IF THERE IS A FLUSH, CHECK THE
    * COMMON CARD TO SEE IF IT MAKES
    * A FLUSH OF 5. *)
   IF RETURN = 4 THEN
   BEGIN
      IF COMMON.RANK <> 0 THEN
        IF COMMON.SUIT = HAND[4].SUIT THEN
          RETURN := 5;
      FLUSH := RETURN
   END
   ELSE
      FLUSH := 0
END; (* FLUSH 7/5/81 *)
(*
 * SCORE
 *)
BEGIN
   RETURN := 0;
   CARDMAX := 4;
   RETURN := RETURN + FLUSH(HAND);  (* CALL FLUSH BEFORE THE
                                       COMMON CARD HAS BEEN
                                       SORTED INTO THE HAND *)

   IF COMMON.RANK <> 0 THEN
   BEGIN
      CARDMAX := 5;
      FOR INDEX := 1 TO PLAYSIZE DO
         IF HAND[INDEX].RANK = JACK THEN
            IF HAND[INDEX].SUIT = COMMON.SUIT THEN
                RETURN := 1;
       HAND[5] := COMMON;
       SORT(5,HAND)
    END
    ELSE
       HAND[5].RANK := 0;
    RETURN := FIFTEEN(HAND) + RETURN;
    RETURN := PAIR(HAND) + RETURN;
    RETURN := RUN(HAND) + RETURN;
    SCORE := RETURN
 END; (* SCORE 7/5/81, D.H. *)
```

## *APPENDIX* A

# Reserved Words and Functions

## *RESERVED WORDS*

The following Pascal reserved words appear in **boldface** in this book:

**AND**
**DIV**
**FILE**
**IN**
**OF**
**RECORD**
**TYPE**
**ARRAY**
**DO**
**FOR**
**LABEL**
**OR**
**REPEAT**
**UNTIL**
**BEGIN**
**DOWNTO**
**FUNCTION**
**MOD**
**PACKED**
**SET**
**VAR**
**CASE**
**ELSE**
**GOTO**
**NIL**
**PROCEDURE**
**THEN**
**WHILE**
**CONST**
**END**
**IF**
**NOT**
**PROGRAM**
**TO**
**WITH**

## *BUILT-IN FUNCTIONS*

| ***Name*** | ***Action*** |
|---|---|
| ABS | Absolute value |
| ARCTAN | Arc tangent |
| CHR | Convert integer to character |
| COS | Cosine |
| EOLN | True when end-of-line is reached |
| EOF | True when end-of-file is reached |

| | |
|---|---|
| EXP | e raised to the power |
| LN | Natural logarithm |
| ODD | True if the integer argument is odd |
| ORD | Convert character to integer |
| PRED | Predecessor or next lower (char, integer or scalar type) |
| ROUND | Convert real to integer by rounding |
| SIN | Sine |
| SQR | Square |
| SQRT | Square root |
| SUCC | Successor or next higher (char, integer or scalar type) |
| TRUNC | Convert real to integer by truncation |

## *UCSD PASCAL PROCEDURES AND FUNCTIONS USED IN THIS BOOK*

| | |
|---|---|
| DELETE | Removes a specified number of characters from a string. In the statement DELETE(STR,BEGIN, NUMBER), STR (a **VAR** parameter) is a string, and BEGIN and NUMBER are integers. NUMBER characters are removed from STR, starting from character BEGIN. |
| EXIT | Terminates a procedure or program. Takes the name of a procedure, function, or program (or the reserved word **PROGRAM**) as its parameter. |
| GOTOXY | Moves the cursor to a position specified by the integer parameters. For example, GOTOXY(25,15) moves the cursor 26 positions across and 16 positions down the screen. (Note that (0,0) is the upper left corner of the screen.) |
| IORESULT | Returns a zero value for an error-free input/output operation. |
| LENGTH | Returns the length of its string parameter. |

| | |
|---|---|
| MARK | Records the state of the heap at the moment MARK is executed. Its parameter is a pointer variable. MARK(P) assigns the first free address of the heap to P. (See RELEASE.) |
| POS | Returns the position of the first character of a given string in another string. Takes two string parameters: POS(FIRST,SECOND) finds the position of the first character of FIRST in SECOND. |
| RELEASE | Restores the heap to its state at the moment procedure MARK was executed. |
| SEEK | Positions the file pointer at a specified place. Takes two parameters: a file and an integer. SEEK (FILENM,POS) moves the pointer to record POS in FILENM. |
| STR | Converts an integer into a string. |

# *APPENDIX* B

# Summary of Pascal

## *MINIMUM STANDARD CHARACTER SET*

| | |
|---|---|
| A full alphabet | A - Z *and/or* a - z |
| The digits | 0 - 9 |
| The special characters | + − * / = < > ( ) [ ] . , ; : ^ |
| A space or blank character | |

## *VARIABLE NAMES*

A Pascal variable name may be any contiguous sequence of alphabetic and numeric characters beginning with a letter. (Some implementations allow certain non-alphanumeric characters, such as the underscore character.)

*Examples:*

CAVE  PLAYER_1  HEARTS  WUMPUSKILLED

The maximum *distinctive* name length varies from one Pascal compiler to the next. Some are limited to a maximum of 6 or 8 characters, while others allow names of 32 characters or more.

## NUMBERS

*Integers* are signed or unsigned strings of digits.

*Examples:*

```
15   −19253   +7   0
```

*Reals* are written with a decimal point, a scale factor or both:

```
379.1275   3.791275E2   3791275E-4
```

The E notation indicates multiplication by powers of 10. Thus, E2 signifies $10^2$. E must always be followed by an integer, signed or unsigned.

## COMMENTS

Comments are typed between the symbols (* and *), and are ignored by the compiler.

*Examples:*

```
(* INTERMEDIATE RESULT *)
(*
 * ROLLDIE.
 * ROLL A DIE, AND PRINT ITS
 * VALUE
 *)
```

On some compilers, the standard symbols { and } may be used as comment delimiters.

## OPERATIONS

### *Integer Operations*

| | |
|---|---|
| + | Addition |
| − | Subtraction |
| * | Multiplication |
| **DIV** | Division (truncated to integer) |

| | |
|---|---|
| **MOD** | Modulo (supplies the *remainder* from the division of two integers) |

### *Real Operations*

| | |
|---|---|
| + | Addition |
| − | Subtraction |
| * | Multiplication |
| / | Division |

### *Boolean Operations*

**AND** **OR** **NOT**

### *Relational Operations*

These operations result in Boolean values TRUE or FALSE.

| | |
|---|---|
| < | Less than |
| > | Greater than |
| = | Equal to |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| <> | Not equal to |
| **IN** | Used with data type **SET**, to determine membership of an element (See "Set Types".) |

## SYNTAX

### *Program Heading*

**PROGRAM** PROGRAM_NAME (FILE_NAME, FILE_NAME,...);

*Example:*

**PROGRAM** BACCARAT (INPUT, OUTPUT)

### *Constant Definition*

**CONST** CONST_NAME = value; CONST_NAME = value;...

*Example:*

```
CONST                    (* FROM BLACKJACK *)
   DECKSIZE       = 52; (* NUMBER OF CARDS IN A DECK *)
   MAXHANDSIZE = 5;   (* MAXIMUM NUMBER OF CARDS *)
```

There are some predefined constants in Pascal:

| | |
|---|---|
| TRUE | Boolean true value |
| FALSE | Boolean false value |
| MAXINT | Largest integer the computer can work with |
| NIL | Null pointer |

Note that a constant must be defined as a simple value, *not* an expression. The second statement below is illegal:

```
CONST
   Y = 3; (*legal*)
   X = Y + 1 (*illegal*)
```

### *Variable Definition*

```
VAR
   VAR_NAME, VAR_NAME,...:TYPE;
   VAR_NAME, VAR_NAME,...:TYPE;...
```

*Example:*

```
VAR       (* FROM NIM *)
   LEFT : INTEGER;
   MYMOVE : BOOLEAN;
```

### *Assignment Statements*

```
VAR_NAME := expression
```

*Example:*

```
(*
 * FROM KENO.
 *)
 I := I + 1;
 MONEY := PAYOFFS + MONEY;
```

### *The Compound Statement*

A compound statement is any sequence of statements bracketed by a **BEGIN/END** pair.

*Example:*

```
(*
 * FROM MATHGAME.
 *)
BEGIN
   WRITE (FIRSTNUM, ' ');
   PRINT := FIRSTNUM * SECNUM;
   WRITE (MULOP); (* OPTIONAL SEMICOLON *)
END
```

### *Procedure Definition*

**PROCEDURE** PROC_NAME (value parameters; **VAR** variable parameters); body of procedure

*Example:*

```
(*
 * WRITE THE SUIT AND VALUE OF A CARD.
 * FROM BLACKJACK.
 *)
PROCEDURE PRINTCARD (ACARD : CARD);
   BEGIN
      WRITE ('THE ', VALUENAME[ACARD.VALUE]);
```

```
        WRITELN (' OF ', SUITNAME[ACARD.SUIT]);
      END;
```

### Function Definition

```
FUNCTION FUNC_NAME (value parameters): result_type;
body of function
```

The value of a function result is assigned by a statement (within the body of the function) of the form

```
FUNCT_NAME := value;
```

*Example:*

```
(*
 * FROM SNAKE.
 * RETURNS A RANDOM INTEGER BETWEEN MIN AND MAX.
 *)
FUNCTION RAND (MIN, MAX : INTEGER) : INTEGER
BEGIN
  RAND := MIN + RANDOM MOD (MAX - MIN + 1);
END;
```

### Placement of Semicolons

Semicolons are required between any two Pascal statements. They are not required either immediately after a **BEGIN** or immediately before an **END**, but their presence there is harmless. A semicolon must *never* immediately precede the **ELSE** in an **IF-THEN-ELSE** construction.

## CONDITIONAL STATEMENTS

### The IF-THEN Statement

```
IF expression THEN statement
```

*Example:*

```
(*
 * FROM TELEPHONE.
 *)
IF (NUMLENGTH < = 0) OR (NUMLENGTH > MAXNUMLENGTH)
    THEN  AREALPHONENUMBER := FALSE;
```

### *The IF-THEN-ELSE Statement*

```
IF expression THEN statement ELSE statement
```

*Example:*

```
(*
 * FROM LIFE.
 *)
IF A >= B THEN
  MAX := A
ELSE
  MAX:= B;
```

### *The CASE Statement*

```
CASE expression OF
  case-labels:statement;
  case-labels:statement;
        .
        .
        .
  case-labels:statement
END
```

*Example:*

```
(*
 * FROM PICTURE.
 * DIRECTION IS A SCALAR VARIABLE.
 * WCHAR IS A TURTLEGRAPHICS PROCEDURE.
 *)
CASE DIRECTION OF
   UP : WCHAR ('^');
   DOWN : WCHAR ('V');
   LEFT : WCHAR ('<');
   RIGHT: WCHAR ('>');
END;
```

## ITERATIVE STATEMENTS

### *The WHILE-DO Statement*

**WHILE** expression **DO** statement

*Example:*

```
(*
 * FROM PASCAL.
 *)
GET(INPUT);
WHILE NOT (INPUT^ IN ['1'..'9'])DO
BEGIN
   READLN(INPUT);
   WRITELN('PLEASE TRY AGAIN');
   WRITE('HOW BIG IS THE TRIANGLE? ');
   GET(INPUT)
END;
```

### *The REPEAT-UNTIL Statement*

**REPEAT** group-of-statements **UNTIL** expression

*Example:*

```
(*
 * FROM WUMPUS.
 * FUNCTION GAMEOVER RETURNS A BOOLEAN VALUE.
 *)
REPEAT
  DOATURN (COMMAND);
UNTIL GAMEOVER;
```

### *The FOR-TO-DO Statement*

```
FOR variable := first-value TO last-value DO statement
```

*Example:*

```
(*
 * FROM BOGGLE.
 * CREATES A RANDOM NUMBER AND CONVERTS IT
 * TO A STRING OF CHARACTERS.
 *)
FOR I := 1 TO NUMDIGITS DO
  NUMBER[1] := CHR(ORD('0') + RANDOM MOD 10);
```

### *The FOR-DOWNTO-DO Statement*

```
FOR variable := first-value DOWNTO last-value DO statement
```

*Example:*

```
FOR I := TERMS DOWNTO 1 DO
  BEGIN
    SUM := 1.0 + I * V/U
    U := SUM
  END;
```

## *TRANSFER-OF-CONTROL STATEMENTS*

It is recommended that **GOTO** statements be avoided except in the rare cases where they actually have some unique value.

### *Label Declaration*

If labels are to be used in a Pascal program they must be explicitly predeclared immediately after the **CONST** declaration section.

```
LABEL integer, integer,...;
```

*Example:*

```
LABEL 99;
```

### *The GOTO Statement*

```
GOTO label;
```

*Example:*

```
BEGIN
   IF ERROR THEN GOTO 99;
                 .
                 .
                 .
   99:
END
```

The **GOTO** in this example causes a clear and orderly termination of the procedure if ERROR is true. The alternative, **IF NOT** ERROR **THEN**, could be harder to manage. Using a **GOTO** statement to jump out of a procedure is questionable programming practice, although some compilers allow it.

## *INPUT AND OUTPUT*

### *Input Procedures*

```
READ(variables);
READLN(variables);
READ(FILE_NAME,variables);
READLN(FILE_NAME,variables);
```

FILE_NAME must be a variable explicitly declared as of type **FILE** or passed as a parameter in the program heading. Note that type TEXT is equivalent to **FILE OF** CHAR.

A READ statement gets values from the current line (strictly speaking, "record"), and assigns them to its variables, leaving any values remaining on the line available for subsequent input.

A READLN statement inputs to its variables from the current line, passing over whatever else may remain on that line. Subsequent input starts from the beginning of the next line.

### *Output Procedures*

```
WRITE(expressions);
WRITELN(expressions);
WRITE(FILE_NAME,expressions);
WRITELN(FILE_NAME,expressions);
```

A WRITE statement simply appends its output to the current output line or record. A WRITELN statement appends its output to the current line, and then issues a carriage return/line feed, i.e., terminates the record.

*Example:*

```
(*
 * FROM OVERUNDER.
 *)
WRITE (' DO YOU WANT INSTRUCTIONS? ');
READLN (CH);
```

### *Formatting Numeric Output*

Each expression in a WRITE or WRITELN can be followed by one or two integers in the following form:

```
integer_expression:integer
real_expression:integer:integer
```

The value of "expression" is output. The first integer is the character width of the output field. The actual value output is right adjusted in this field and preceded by blanks. All the digits of the value will always be output even if this output overflows the field width.

In the case of a real number, the second integer defines the number of digits to be output to the right of the decimal point.

*Example:*

```
(*
 * FROM KENO
 *)
WRITELN ('YOUR TOTAL MONEY IS $', MONEY:4:2);
```

## DATA TYPES

**TYPE** declarations follow the **CONST** declarations and precede the **VAR** declarations.

### Scalar Types

```
TYPE TYPE_NAME = (identifier, identifier,...);
```

*Example:*

```
(*
 * FROM DODGE.
 *)
TYPE
  WHAT = (ME, ATTACKER, TANK, MINE, NOTHING);
  DIRECTION = (STAY,N,NE,E,SE,S,SW,W,NW,NOWHERE);
```

### Subrange Types

```
TYPE
  TYPE_NAME = constant..constant;
VAR
  VAR_NAMES : constant..constant;
```

*Example:*

```
(*
 * FROM TWINKLE.
 *)
INDEX = 1..SCREENSIZE;
```

### *Array Types*

```
TYPE
  TYPE_NAME = ARRAY[index_type,index_type,...]
                OF type;
VAR
  VAR_NAMES : ARRAY[index_type, index_type,...]
                OF type;
```

A multidimensional array requires that an index _ type be specified for each dimension. An index _ type is usually a subrange type but may be any scalar type.

*Examples:*

```
(*
 * FROM CRIBBAGE.
 *)
TYPE
  HANDTYPE = ARRAY[1..DEALSIZE] OF CARD;
VAR
  COMP : HANDTYPE;
  HUMAN : HANDTYPE;
  CRIB : HANDTYPE;


(*
 * FROM HORSERACE.
 *)
VAR
  TRACK : ARRAY[1..MAXPLAYER, 1..MAXHORSES] OF HORSE;
```

### *Referencing Array Elements*

Array indices appear between square brackets. In the case of multidimensional arrays, the index expressions are separated by commas. The subscript index for each dimension may be any expression that evaluates to the declared index type.

```
ARRAY_NAME [index,index,...]
```

*Example:*

```
(*
 * FROM CRIBBAGE.
 *)
HUMAN[CARDNUM] := HUMAN[CARDNUM + 1];
```

### *Packed Arrays*

In some implementations the amount of memory space used by an array may be reduced by declaring the array as **PACKED.** The actual amount of space saved and the resulting slowdown in access time depend on the implementation.

```
TYPE
   TYPE_NAME = PACKED ARRAY [index_type,
                        index_type...]OF element_type;
VAR
   VAR_NAME: PACKED ARRAY [index_type,
                      index_type...] OF element_type;
```

*Example:*

```
TYPE
   LINE_IMAGE = PACKED ARRAY [1...MAXCOL] OF CHAR
```

Note that UCSD Pascal has the type STRING, which is essentially a **PACKED ARRAY OF** CHAR with a "dynamic" length.

### *Record Types*

```
TYPE
   TYPE_NAME = RECORD field-list END;
```

```
VAR
  VAR_NAME: RECORD field-list END;
```

*Example:*

```
(*
 * FROM LIFE.
 *)
TYPE
  EDGES = RECORD
        LEFT,
        RIGHT,
        TOP, BOTTOM : INTEGER;
     END;
VAR
  EDGE : EDGES;
```

The fields of *variant* records may change from record to record. The different structures that may occur in the variant portion of the record are specified using a **CASE**-like construction that *must* follow the declaration of the *fixed* fields of the record.

*Example:*

```
TYPE
  DATE = RECORD
    MO: 1..12;
    YR: INTEGER
  END(*date*);
  GRADE = (FRESH, SOPH, JR, SEN);
  STUDENT = RECORD
    NAME: PACKED ARRAY [1..20] OF CHAR;
    NUM: INTEGER;
    CASE GR:GRADE OF
      FRESH, SOPH: (MAJOR: BOOLEAN);
      JUN: (CREDITS: INTEGER);
      SEN: (GDATE: DATE)
  END(*student*);
```

### *Referencing Record Fields*

A field within a record variable is referenced by the so-called "dot" notation; the field name (which appears in the record type declaration) is joined with a dot to the actual record variable name:

```
RECORD_NAME.FIELD_NAME
```

*Examples:*

```
(*
 * FROM LIFE.
 *)
EDGE.TOP := HEIGHT - 1;
EDGE.RIGHT := MINBOUND + 1;
EDGE.LEFT := WIDTH - 1;
EDGE.BOTTOM := MINBOUND + 1;
```

Where many accessses are to be made to fields within the same record variable(s) a context may be established using the **WITH** statement:

```
WITH RECORD_NAME DO statement
```

*Example:*

```
(*
 * FROM LIFE.
 *)
WITH EDGE DO
BEGIN
  LEFT := MIN(LEFT, X);
  RIGHT := MAX(RIGHT, X);
  TOP := MIN(TOP, Y);
  BOTTOM := MAX(BOTTOM, Y);
END;
```

The **WITH** construct makes the program clearer to read and easier to compile efficiently.

### *Set Types*

```
TYPE
  TYPE_NAME = SET OF base_type;
VAR
  VAR_NAME: SET OF base_type;
```

*Example:*

```
(*
 * FROM WUMPUS.
 *)
TYPE
  ROOM = 1..MAXROOMS;
  ROOMS = SET OF ROOM;
VAR
  CAVE : ARRAY[ROOM] OF ROOMS;
  BATS : ROOMS;
  PITS : ROOMS;
```

### *Set Operations*

| | |
|---|---|
| + | Union |
| * | Intersection |
| − | Difference |
| = | Set equality |
| <> | Set inequality |
| **IN** | Set membership. TRUE if the scalar operand on the left is an element of the set operand on the right. |
| <= | Set inclusion. TRUE if all elements **IN** the left operand are also **IN** the right operand. |
| >= | Set containment. TRUE if all elements **IN** the right operand are also **IN** the left operand. |

*Examples:*

```
(*
 * FROM WUMPUS.
 *)
IF (PLAYER IN CAVE[WUMPUS]) OR
   ((CAVE[PLAYER] * CAVE[WUMPUS]) <> [ ]) THEN
      WRITELN('I SMELL A WUMPUS.');
         . . .
REPEAT
      FLEWTO := RAND(1, MAXROOMS)
UNTIL (NOT (FLEWTO IN (BATS+PITS))) AND
      (FLEWTO <> WUMPUS);
WRITELN('A SUPERBAT PICKED YOU UP AND CARRIED YOU OFF.');
```

### *File Types*

```
TYPE
   FILE_NAME = FILE OF type;
VAR
   VAR_NAME: FILE OF type;
```

*Example:*

```
(*
 * FROM HANGMAN.
 *)
TYPE
   WORD = STRING[STRINGSIZE];
VAR
   DICT: FILE OF WORD;
```

The standard I/O procedures for use with files are:

| | |
|---|---|
| RESET | Opens a file so that it can be read. |
| REWRITE | Opens a file so that it can be written (all components previously written on the file are lost). |

| | |
|---|---|
| GET | Transfers one component of a file to the associated buffer variable. |
| PUT | Appends the contents of a buffer variable to its file. |

### *Pointer Types*

A pointer variable contains the address of a *dynamic variable*. Dynamic variables are variables that are created during program execution by the procedure NEW. Such variables are referenced indirectly by pointer variables. A pointer variable can be initialized to the value **NIL**, meaning the pointer is not pointing to any dynamic variable.

*Example:*

```
(*
 * FROM PASCAL.
 *)
TYPE
  NODEPTR = ^NODE;
  NODE = RECORD
    VALUE : INTEGER;
    NEXT : NODEPTR
  END;
VAR
  START : NODEPTR;
  CURNODE:NODEPTR;
        . . .
  NEW(CURNODE);
  CURNODE^.VALUE := 1;
  CURNODE^.NEXT := START;
  START := CURNODE
```

# *The SYBEX Library*

### BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS
**by Alan R. Miller** 340 pp., 120 illustr., Ref. B240

This second book in the "Programs for Scientists and Engineers" series provides a library of problem solving programs while developing proficiency in BASIC.

### INSIDE BASIC GAMES
**by Richard Mateosian** 350 pp., 240 Illustr., Ref. B245

Teaches interactive BASIC programming through games. Games are written in Microsoft BASIC and can run on the TRS-80, APPLE II and PET/CBM.

### FIFTY BASIC EXERCISES
**by J.P. Lamoitier** 240 pp., 195 Illustr., Ref. B250

Teaches BASIC by actual practice, using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.

### EXECUTIVE PLANNING WITH BASIC
**by X.T. Bui** 192 pp., 19 illustr., Ref. B380

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

### BASIC FOR BUSINESS
**by Douglas Hergert** 250 pp., 15 illustr., Ref. B390

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully explained accounting programs, and shows you how to write them.

### BASIC EXERCISES FOR THE APPLE
**by J.P. Lamoitier** 230 pp., 80 illustr., Ref. B500

This book is an Apple version of *Fifty BASIC Exercises.*

### YOUR FIRST COMPUTER
**by Rodnay Zaks** 260 pp., 150 Illustr., Ref. C200A

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

### DON'T (or How to Care for Your Computer)
**by Rodnay Zaks** 220 pp., 100 Illustr., Ref. C400

The correct way to handle and care for all elements of a computer system, including what to do when something doesn't work.

### INTRODUCTION TO WORD PROCESSING
**by Hal Glatzer** 200 pp., 70 illustr., Ref. W101

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

### INTRODUCTION TO WORDSTAR
**by Arthur Naiman** 200 pp., 30 illustr., Ref. W105

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

### FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

**by Rodnay Zaks** 560 pp., 255 illustr., Ref. C201A

A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

### MICROPROCESSOR INTERFACING TECHNIQUES

**by Rodnay Zaks and Austin Lesea** 460 pp., 400 Illustr., Ref. C207

Complete hardware and software interconnect techniques including D to A conversion, peripherals, standard buses and troubleshooting.

### PROGRAMMING THE 6502

**by Rodnay Zaks** 390 pp., 160 Illustr., Ref. C202

Assembly language programming for the 6502, from basic concepts to advanced data structures.

### 6502 APPLICATIONS BOOK

**by Rodnay Zaks** 280 pp., 205 Illustr., Ref. D302

Real life application techniques: the input/output book for the 6502.

### ADVANCED 6502 PROGRAMMING

**by Rodnay Zaks** 300 pp., 140 Illustr., Ref. G402A

Third in the 6502 series. Teaches more advanced programming techniques, using games as a framework for learning.

### PROGRAMMING THE Z80

**by Rodnay Zaks** 620 pp., 200 Illustr., Ref. C280

A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

### PROGRAMMING THE Z8000

**by Richard Mateosian** 300 pp., 125 Illustr., Ref. C281

How to program the Z8000 16-bit microprocessor. Includes a description of the architecture and function of the Z8000 and its family of support chips.

### THE CP/M HANDBOOK (with MP/M)

**by Rodnay Zaks** 330 pp., 100 Illustr., Ref. C300

An indispensable reference and guide to CP/M—the most widely used operating system for small computers.

### INTRODUCTION TO PASCAL (Including UCSD PASCAL)

**by Rodnay Zaks** 420 pp., 130 Illustr., Ref. P310

A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

### THE PASCAL HANDBOOK

**by Jacques Tiberghien** 490 pp., 350 Illustr., Ref. P320

A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

### PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS

**by Alan R. Miller** 400 pp., 80 Illustr., Ref. P340

A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes such programs as curve-fitting, integrals and statistical techniques.

### APPLE PASCAL GAMES

**by Douglas Hergert and Joseph T. Kalash** 380 pp., 40 illustr., Ref. P360

A collection of the most popular computer games in Pascal challenging the reader not only to play but to investigate how games are implemented on the computer.

### INTRODUCTION TO THE UCSD p-SYSTEM

**by Charles W. Grant and Jon Butah** 320 pp., 110 illustr., Ref. P370

A simple, clear introduction to the UCSD Pascal Operating System for beginners through experienced programmers.

### INTERNATIONAL MICROCOMPUTER DICTIONARY

140 pp., Ref. X2

All the definitions and acronyms of microcomputer jargon defined in a handy pocket-size edition. Includes translations of the most popular terms into ten languages.

### MICROPROGRAMMED APL IMPLEMENTATION

**by Rodnay Zaks** 350 pp., Ref. Z10

An expert-level text presenting the complete conceptual analysis and design of an APL interpreter, and actual listings of the microcode.

### MASTERING CP/M

**Alan R. Miller** 320 pp., C302

For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system . . . takes up where our *CP/M Handbook* leaves off.

### THE APPLE CONNECTION

**James W. Coffron** 228 pp., C405

Teaches elementary interfacing and BASIC programming of the Apple for connection to external devices and household appliances.

### BASIC EXERCISES FOR THE IBM PERSONAL COMPUTER

**J.P. Lamoitier** 232 pp., B510

This book is an IBM version of *Fifty BASIC Exercises*.

### FORTRAN PROGRAMS FOR SCIENTISTS AND ENGINEERS

**Alan R. Miller** 320 pp., F440

Third in the "Programs for Scientists and Engineers" series. Specific scientific and engineering application programs written in FORTRAN.

### PROGRAMMING THE 6809

**Rodnay Zaks & William Labiak** 520 pp., C209

This book explains how to program the 6809 in assembly language. No prior programming knowledge required.

### CELESTIAL BASIC: Astronomy on Your Computer

**Eric Burgess** 228 pp., B330

A collection of BASIC programs that rapidly complete the chore of typical astronomical computations. It's like having a planetarium in your own home! Displays apparent movement of stars, planets and meteor showers.

# SYBEX® COMPUTERBOOKS ARE DIFFERENT.

## Here is why . . .

At SYBEX, each book is designed with you in mind. Every manuscript is carefully selected and supervised by our editors, who are themselves computer experts. Programs are thoroughly tested for accuracy by our technical staff. Our computerized production department goes to great lengths to make sure that each book is designed as well as it is written. We publish the finest authors, whose technical expertise is matched by an ability to write clearly and to communicate effectively.

In the pursuit of timeliness, SYBEX has achieved many publishing firsts. SYBEX was among the first to integrate personal computers used by authors and staff into the publishing process. SYBEX was the first to publish books on the CP/M operating system, microprocessor interfacing techniques, word processing, and many more topics.

Expertise in computers and dedication to the highest quality in book publishing have made SYBEX a world leader in microcomputer education. Translated into fourteen languages, SYBEX books have helped millions of people around the world to get the most from their computers. We hope we have helped you, too.

# FOR A COMPLETE CATALOG OF OUR PUBLICATIONS

U.S.A.
2344 Sixth Street
Berkeley,
California 94710
Tel: (415) 848-8233
Telex: 336311

SYBEX-EUROPE
4 Place Félix Eboué
75583 Paris Cedex 12
Tel: 1/347-30-20
Telex: 211801

SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
West Germany
Tel: (0211) 287066
Telex: 08 588 163

SYBEX®

ENTER THE WORLD OF

# Apple Pascal Games

EXPAND YOUR REPERTOIRE OF GAMES AND LEARN

GUESSIT, KENO, OVERUNDER, TWINKLE, MATHGAME, CHUCKALUCK, GUNSHOT, HORSERACE, KISMET, BLACKJACK, BLOCK, BACCARAT, TELEPHONE, SNAKE, DODGE, NIM, NUMCONVERT, ARTILLERY, PASCAL, BOGGLE, WUMPUS, BLACKBOX, PICTURE, CRIBBAGE, LANDER, HANGMAN, LIFE

## EXPLORE

all the essential elements of UCSD Pascal, and learn the important Apple Pascal extensions.

## EXPERIENCE

each game with:

- a description of the game
- a sample run
- a "structure chart"
- an explanation of the Pascal program
- a program listing

## ENJOY

the games, and learn Pascal in the process.

## THE AUTHORS

*Douglas Hergert* is a member of the Sybex editorial staff, and has contributed to Sybex books as editor, writer and translator.

*Joseph T. Kalash* has written and developed a large number of Pascal programs at Sybex, including many of those presented in this book.

ISBN 0-89588-074-1

**Hergert**

**Kalash**

# Apple Pascal Games

SYBEX

0-074